



NeXTstep Concepts

Important Information

This version of the *NeXTstep® Concepts* manual is based on Release 1.0. It's included in the current release of the *Developer's Library* as a temporary means of providing you with necessary conceptual information about NeXTstep. As described below, specific chapters in this manual will be replaced by updated concepts material. Until that material is available, this manual—when taken in the context of the release notes—can help give you an overview of NeXTstep.

Release Notes

The current NeXTstep programming environment differs in various ways from the 1.0 environment described in this manual. Generally, these differences are in the form of additional functionality, although there have also been some isolated, but significant, changes to existing systems and to the user interface. The release notes are your source of information about the changes between the 1.0 release and the current release of NeXTstep.

You can access the release notes either from the Digital Librarian™ or directly from their location in **/NextLibrary/Documentation/NextDev/ReleaseNotes**. The release notes that are the most relevant to the subjects presented in this manual are:

- AppKit.rtf
- WindowServer.rtf
- AllocInitAndNew.rtf

In addition, you'll find updated information on user interface guidelines and other subjects in **/NextLibrary/Documentation/NextDev/Notes**.

NeXTstep Concepts Updates

In an effort to provide you with updated material in a timely manner, specific chapters of this manual will be replaced by a series of shorter manuals. These shorter manuals will be made available through developer mailings.



NeXT Developer's Library

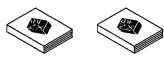
NeXTstep

Draw upon the library of software contained in NeXTstep to develop your applications. Integral to this development environment are the Application Kit and Display PostScript.



Concepts

A presentation of the principles that define NeXTstep, including user interface design, object-oriented programming, event handling, and other fundamentals.



Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

Let your application listen, talk, and sing by using the Sound Kit and the Music Kit. Behind these capabilities is the DSP56001 digital signal processor. Independent of sound and music, scientific applications can take advantage of the speed of the DSP.



Concepts

An examination of the design of the sound and music software, including chapters on the use of the DSP for other, nonaudio uses.



Reference

Detailed, comprehensive descriptions of each piece of the sound, music, and DSP software.



NeXT Development Tools

A description of the tools used in developing a NeXT application, including the Edit application, the compiler and debugger, and some performance tools.



NeXT Operating System Software

A description of NeXT's operating system, Mach. In addition, other low-level software is discussed.



Writing Loadable Kernel Servers

How to write loadable kernel servers, such as device drivers and network protocols.



NeXT Technical Summaries

Brief summaries of reference information related to NeXTstep, sound, music, and Mach, plus a glossary and indexes.



Supplemental Documentation

Information about PostScript, RTF, and other file formats useful to application developers.



NeXTstep Concepts



We at NeXT Computer have tried to make the information contained in this manual as accurate and reliable as possible. Nevertheless, NeXT disclaims any warranty of any kind, whether express or implied, as to any matter whatsoever relating to this manual, including without limitation the merchantability or fitness for any particular purpose. NeXT will from time to time revise the software described in this manual and reserves the right to make such changes without obligation to notify the purchaser. In no event shall NeXT be liable for any indirect, special, incidental, or consequential damages arising out of purchase or use of this manual or the information contained herein.

Copyright ©1990 by NeXT Computer, Inc. All Rights Reserved.
[2908.00]

The NeXT logo and NeXTstep are registered trademarks of NeXT Computer, Inc., in the U.S. and other countries. NeXT, AppInspector, Digital Librarian, Digital Webster, Interface Builder, Music Kit, Sound Kit, and Workspace Manager are trademarks of NeXT Computer, Inc. Display PostScript and PostScript are registered trademarks of Adobe Systems Incorporated. UNIX is a registered trademark of AT&T. Helvetica and Times are registered trademarks of Linotype AG and/or its subsidiaries and are used herein pursuant to license. Wreath and Monogram is a registered trademark of Merriam-Webster, Incorporated and is used herein pursuant to license. WriteNow is a registered trademark of T/Maker Company. Mathematica is a registered trademark of Wolfram Research, Inc. All other trademarks mentioned belong to their respective owners.

Notice to U.S. Government End Users:

Restricted Rights Legends

For civilian agencies: This software is licensed only with "Restricted Rights" and use, reproduction, or disclosure is subject to restrictions set forth in subparagraph (a) through (d) of the Commercial Computer Software—Restricted Rights clause at 52.227-19 of the Federal Acquisition Regulations.

Unpublished—rights reserved under the copyright laws of the United States and other countries.

For units of the Department of Defense: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

NeXT Computer, Inc., 900 Chesapeake Drive, Redwood City, CA 94063.

Manual written by Don Larkin, Matt Morse, Jackie Neider, and Caroline Rose

Edited by Caroline Rose

Book design by Eddie Lee

Illustrations by Jeff Yaksick and Don Donoughe

Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs

Publications management by Cathy Novak

Reorder Product #N6007B

Contents

1-1 Chapter 1: System Overview

- 1-5 NeXTstep
- 1-8 The Mach Operating System
- 1-9 Sound and Music Overview

2-1 Chapter 2: The NeXT User Interface

- 2-5 Design Philosophy
- 2-11 User Actions
- 2-30 The Interface to the Operating System
- 2-37 The Window Interface to Applications
- 2-52 Application and Window Status
- 2-60 Menus
- 2-75 Panels
- 2-81 Controls

3-1 Chapter 3: Object-Oriented Programming and Objective-C

- 3-3 Objects
- 3-4 Messages
- 3-7 Classes
- 3-16 How Messaging Works
- 3-25 The Object Class
- 3-29 Options
- 3-34 Type Encoding
- 3-36 Language Synopsis

4-1 Chapter 4: Drawing

- 4-4 Design Philosophy
- 4-6 The Screen
- 4-19 The Window System
- 4-29 Compositing and Transparency
- 4-45 Instance Drawing
- 4-47 Sending PostScript Code to the Window Server
- 4-51 Imaging Conventions

5-1 Chapter 5: Events

- 5-4 Event Basics
- 5-4 Types of Events
- 5-9 The Event Record
- 5-17 Keyboard Information
- 5-18 Event Masks
- 5-20 The Event Queue
- 5-21 Event-Related Services

6-1 Chapter 6: Program Structure

- 6-4 Writing a Program with the Application Kit
- 6-12 Principal Application Kit Classes
- 6-37 Program Framework
- 6-50 Managing Windows
- 6-63 Environmental Information
- 6-68 Application Kit Conventions

7-1 Chapter 7: Program Dynamics

- 7-5 Event Handling
- 7-69 Drawing in the View Hierarchy
- 7-106 Printing

8-1 Chapter 8: Interface Builder

- 8-6 Interface Builder and Program Design
- 8-11 Interface Builder Tutorial
- 8-53 Interface Builder Reference

9-1 Chapter 9: User-Interface Objects

- 9-4 The Text Class
- 9-35 The Box Class

10-1 Chapter 10: Support Objects and Functions

- 10-4 Streams
- 10-11 Archiving to a Typed Stream
- 10-21 The Defaults System
- 10-31 The Pasteboard
- 10-36 Exception Handling

Index

Chapter 1

System Overview

1-5 NeXTstep

- 1-5 Interface Builder
- 1-6 The Application Kit
- 1-6 The NeXT Window Server
- 1-7 Drawing with Display PostScript
- 1-7 Handling Events

1-8 The Mach Operating System

1-9 Sound and Music Overview

- 1-9 Sound Kit
- 1-10 Music Kit
- 1-10 Creating and Storing Music Data
- 1-10 Creating and Performing Musical Sounds

Chapter 1

System Overview

As illustrated in Figure 1-1, there are four levels of software between a NeXTTM application program and the hardware that executes it:

- The NeXT Interface BuilderTM
- Objective-C software “kits”
- The NeXT Window Server and specialized C libraries
- The Mach operating system

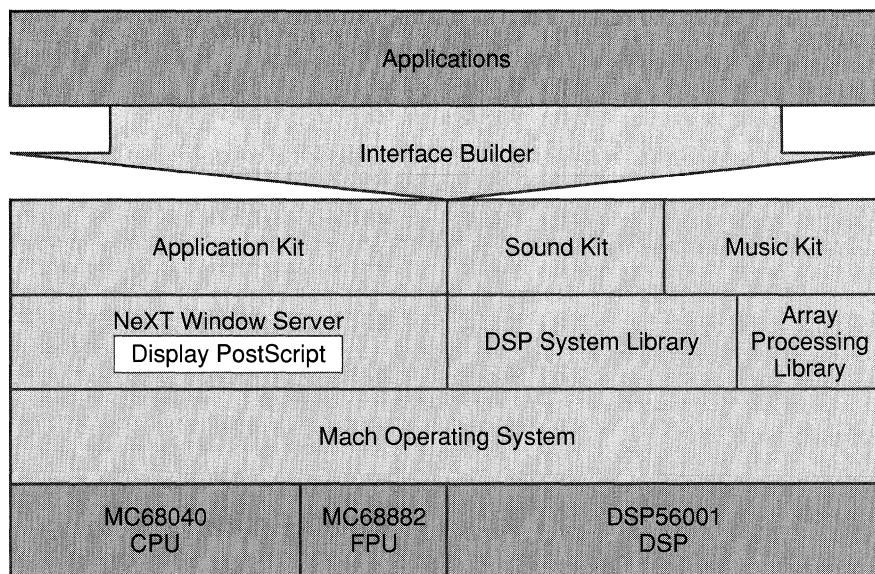


Figure 1-1. System Overview

Interface Builder is a powerful tool that lets you graphically design your application’s user interface. It also makes it easy for you to establish connections between user-interface objects and your own code (for example, the code to execute when a button on the screen is clicked).

NeXT application programs are written in Objective-C, an extension to C that adds object-oriented concepts to the language. The software kits define a number of *classes*, or object templates, that you can use in your own applications. The software kits currently provided by NeXT are:

- An *Application Kit* that every application uses to implement the NeXT window-based user interface
- *Sound Kit*TM for adding sounds to your application, doing speech analysis, and performing other sound manipulation
- *Music Kit*TM for music composition, synthesis, and performance

The *NeXT Window Server* is a low-level background *process* used by the Application Kit to manage windows and to send user *events*, such as mouse and keyboard actions, back to an application. Included in the Window Server is a Display PostScript[®] interpreter that's used for all drawing of text and graphics on the screen or printed page. The Display Postscript system was jointly developed by NeXT and Adobe Systems Inc. as an enhancement of Adobe's PostScript[®] page description language.

Sound Kit and Music Kit use the DSP56001 digital signal processor (the *DSP*) as a sound synthesizer. Objects in these kits communicate with the DSP by calling functions in the DSP system library. In addition to establishing and managing a channel of communication between your application and the DSP, the functions in the DSP system library also provide diagnostic capabilities and data conversion routines.

The functions in the array processing library use the DSP as an array processor, allowing your application to process multidimensional data with great speed and efficiency. Any application can include and use the array processing library.

Mach is a multitasking operating system developed at Carnegie Mellon University. It acts as an interface between the upper levels of software and the three Motorola microprocessors provided with the NeXT computer: the MC68040 central processor, the MC68882 floating-point coprocessor, and the DSP56001 digital signal processor.

The rest of this chapter elaborates on this simplified overview. In the next section, Interface Builder, the Application Kit, and the Window Server are described as part of the *NeXTstep* working environment. Subsequent sections describe Mach and the sound and music facilities, which aren't included in NeXTstep. From this base of knowledge about the NeXT system, you can go on to read the chapters that address your areas of interest.

NeXTstep

NeXTstep combines the essential components of the software design into a working environment for both the user and the application developer. Figure 1-2 shows the software elements of NeXTstep.

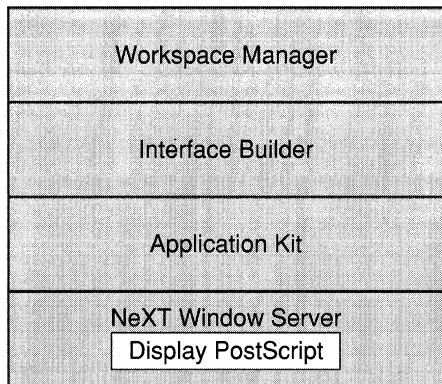


Figure 1-2. NeXTstep

Workspace Manager™ is an application that runs automatically after you log in. From Workspace Manager you can launch applications, manage files, and perform other tasks, as described in detail in the *NeXT User's Reference* manual.

Interface Builder

Interface Builder is a powerful application that has a twofold purpose:

- It lets you graphically design a user interface for your application.
- It creates a programming environment for each new project.

Interface Builder displays a window that represents your application's user interface and provides graphic access to a number of standard interface objects, such as text fields, buttons, and menus. To design an interface, you simply drag the interface objects into your application's interface window and place them where you want them. You can inspect and modify an object to create a particular look—you can even give it a sound effect.

Interface objects understand user events such as mouse and keyboard actions and automatically perform basic display operations when an action is directed at them. For example, a button knows how to graphically highlight itself when the user clicks it, a text field displays the characters that the user types in it, a window disappears when its close button is clicked, and so on.

In addition, Interface Builder has tools for connecting interface objects to each other. For instance, you can connect a button to a panel such that the panel comes to the front when the user clicks the button, or you can connect a slider to a text field so that the value displayed in the text field is continuously updated as the user drags the slider's knob up and down. NeXT provides the code for the basic interface operations; you can also design your own objects and actions and connect them in Interface Builder. For some simple applications, the interface objects and the code provided by NeXT are sufficient, allowing you to create an application without writing a line of code.

Most applications, of course, require more work than simply defining an interface. The other facet of Interface Builder, its creation of a programming environment, makes it a good place to start a new project even if the interface is trivial compared to the amount of programming the project requires. Interface Builder can automatically create a UNIX® *makefile* (the script for your application's compilation routine), some basic source code, and the header files that your application needs to compile.

The Application Kit

All applications use the Application Kit regardless of their purpose and complexity. The buttons, sliders, and windows that you use to design an interface with Interface Builder are defined as classes in the Application Kit. Also, as described in the next section, it's through this Kit that your application is able to draw on the screen and receive events from the user.

The Objective-C language and the software kits make it easy to create your own class of object. One of the features of the language is that it supports class inheritance; this means that you can create a class that inherits the attributes of another class. For example, you can create a class that inherits from the Application Kit's Button class (by convention, class names are capitalized in Objective-C). Your version of Button will be able to do everything that the Kit version can do, plus you can add to it the specialized functionality that your application requires.

The NeXT Window Server

The NeXT Window Server is a low-level background process that creates and manipulates windows on the screen. Your application establishes a connection with the Window Server through the Application Kit and opens one or more windows. Windows provide a vehicle for communication between the user and the application. The Window Server manages this communication as it fulfills two functions:

- It draws images on the screen according to instructions sent from your application.
- It sends user events back to your application.

Drawing with Display PostScript

The Window Server draws images with NeXT's implementation of the Display PostScript system. Display Postscript provides an interactive, display-oriented environment that's independent of any window system. NeXT's implementation extends the language with features unique to the NeXT window system.

All the Display PostScript operators and the NeXT extensions to the language can be accessed as C functions. In addition, NeXT supplies a program named **pswrap** that lets you generate C functions that correspond to your own PostScript procedures.

Handling Events

Besides drawing images on the screen, the Window Server also identifies user events and dispatches them to your application. Through a mechanism defined in the Application Kit, the event is forwarded to the appropriate object:

- The event may be handled entirely by an Application Kit object. For example, if the user chooses a command that edits the text of a Text object, the operation is handled entirely by code that's built into the definition of the Text class in the Application Kit.
- The Application Kit object may do some of the event handling, leaving the rest to your code. If, for example, the user clicks a button on the screen, the Application Kit's definition of the Button class takes care of highlighting and unhighlighting the button, while your code performs application-specific actions associated with the object.

Figure 1-3 shows the overall data flow for a typical application that accepts input from the keyboard and mouse and displays output on the screen.

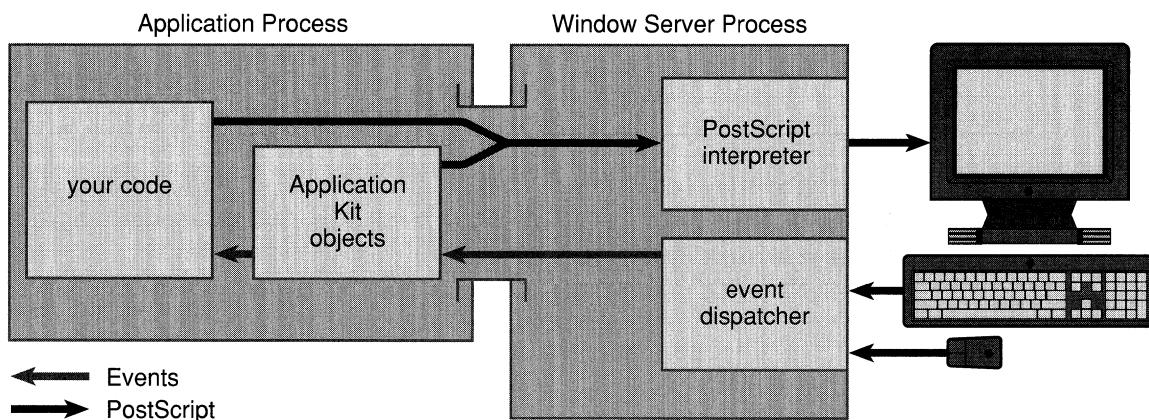


Figure 1-3. Window System Data Flow

In this program model, the application consumes user events and draws on the screen. This largely replaces the UNIX model in which an application reads from the standard input stream and writes to the standard output stream. The Application Kit and the Window Server work together to reduce the work you must do to build applications that interact with the user.

The Mach Operating System

The Mach operating system provides complete compatibility with UNIX 4.3BSD (Berkeley Software Distribution) but adds a faster and more consistent system of interprocess communication, a larger virtual memory space, memory-mapped files, and multiple threads of execution within a single address space. Mach gives programmers the entire standard UNIX environment; existing machine-independent UNIX 4.3BSD applications need only be recompiled to run on the NeXT computer.

Every running application is a separate process. In Mach, several processes may be running concurrently. For example, the Window Server process runs at the same time as all currently executing application processes (see Figure 1-4).

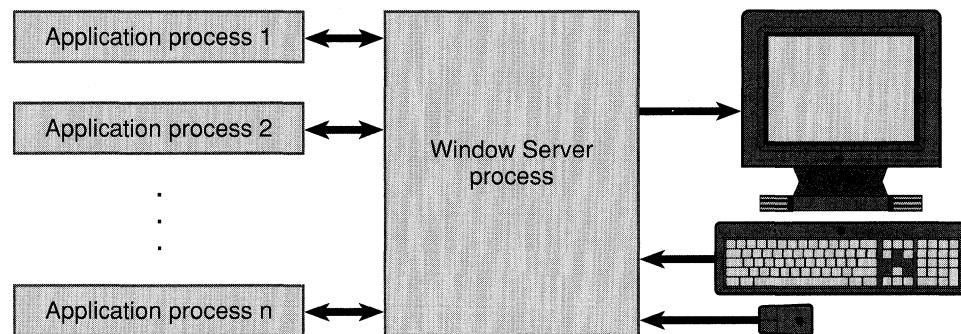


Figure 1-4. Window Server and Application Processes

In addition to providing a multitasking environment, Mach lets processes communicate with each other. This feature is used by applications such as Digital Webster™, which can look up a word selected from text that's displayed by an entirely different application.

Sound and Music Overview

The NeXT computer provides a powerful system for creating and manipulating sound and music. The software for this system is divided into two kits: Sound Kit and Music Kit. The kit that you need depends on the demands of your application:

- Sound Kit lets you incorporate prerecorded sound effects into your application and provides easy access to the microphone input so you can record your own sounds. The objects in Sound Kit let you examine and manipulate sound data with microscopic precision.
- Music Kit provides tools for composing, storing, and performing music. It lets you communicate with external synthesizers as well as create your own software instruments. Like Sound Kit, Music Kit provides objects that create and manipulate sounds with exquisite detail, but, more importantly, Music Kit helps you organize and arrange groups of sounds and design a performance.

Sound Kit

A small number of system beep-type sound recordings, stored in files on the disk (called *soundfiles*), are provided by NeXT. Through Sound Kit, you can easily access these files and incorporate the sounds into your application. It's also extremely easy to record new sounds into the NeXT computer. Simply plug a microphone into the input jack at the back of the monitor and, with a single message to the Sound Kit's Sound object, you can record your own sound effect. Sound playback is just as simple: another message and the sound is played on the internal speaker and sent to the stereo output jacks at the back of the monitor.

When you record a sound using the Sound object, a series of audio "snapshots" or *samples* is created. By storing sound as samples, you can analyze and manipulate your sound data with an almost unlimited degree of precision. The SoundView class lets you see your sounds by displaying the samples in a window.

While Sound Kit is designed primarily for use on sampled data, you can also use it to send instructions to the DSP. The speed of the DSP makes it an ideal sound synthesizer and, in general, DSP instructions take up much less space than sampled data. The Sound object manages the details of playing sounds for you, so you needn't be aware of whether a particular Sound contains samples or DSP instructions.

Music Kit

Music Kit provides a number of ways to compose and perform music. By attaching an external synthesizer keyboard to a serial port, you can play the NeXT computer as a musical instrument. Alternatively, you can compose music to be played by the computer by creating music data in a text editor or by creating an algorithm that generates it automatically. These approaches can be combined in performance. For instance, a musician can use an external keyboard to trigger precomposed events, allowing the computer to create sounds and gestures that are impossible on a traditional instrument, but at moments specified by the performer.

Creating and Storing Music Data

Music Kit represents music as a series of Note objects. Each Note object describes the characteristics of a musical note, such as its pitch, loudness, and duration. How a Note object is performed depends on the detail it contains.

Notes can be stored in a file as statements written in *ScoreFile*, a music language developed at NeXT that represents music as text. A file written in the ScoreFile language is called a *scorefile*. In addition to reading and writing scorefiles from an application, you can also create and modify them with a text editor.

Music Kit recognizes the *MIDI* (Musical Instrument Digital Interface) standard. You can attach a MIDI instrument to a serial port at the back of the computer and capture a performance. MIDI commands are turned into Note objects that can be manipulated and stored.

Creating and Performing Musical Sounds

As mentioned earlier, one of the benefits of the DSP56001 is that it can be used to synthesize sounds. The generality of the DSP allows a wide range of synthesis techniques; in fact, the DSP can emulate almost any commercially available keyboard synthesizer. A number of ready-to-use DSP software instruments are provided as Objective-C classes in the Music Kit. Software instruments are constructed from synthesis building blocks also written in Objective-C, so you can easily modify the existing instruments or design your own.

Just as you can enter music data through MIDI, you can perform music on an external synthesizer by sending MIDI data back out a serial port. Music Kit and music-related device drivers are designed to handle synchronization for you, allowing you to synthesize music on the DSP and send MIDI data to an external synthesizer at the same time.

By using objects from both Music Kit and Sound Kit, you can create an instrument that plays sampled data. For instance, you can use a Sound object to record a single tone from a traditional instrument and then play the tone back at the pitches and times specified by a series of Note objects. You can also use Sound Kit to record, in soundfiles, entire musical performances synthesized on the DSP.

Figure 1-5 shows the components for creating, playing, and storing music and sound with the hardware and software of the NeXT computer.

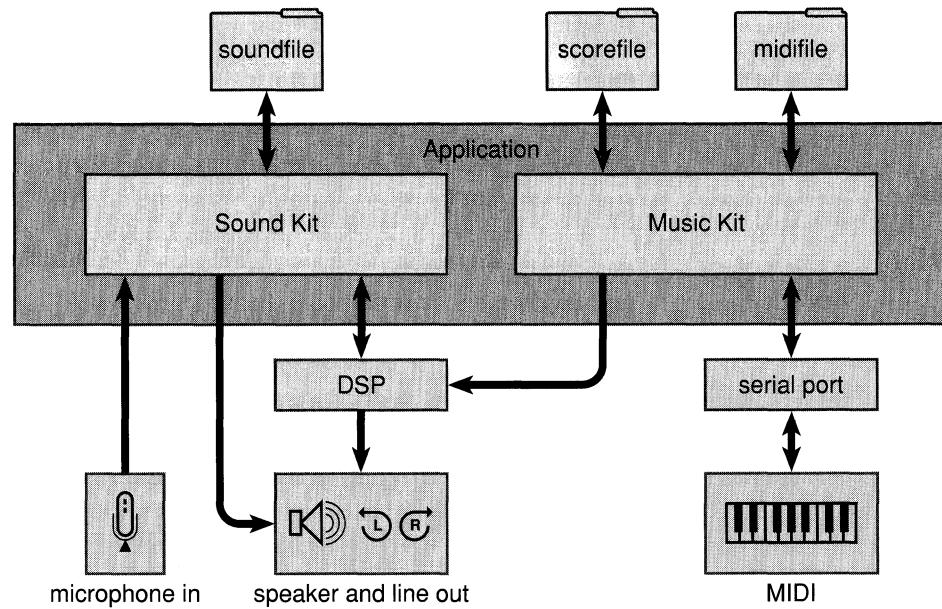


Figure 1-5. Music and Sound Components

Chapter 2

The NeXT User Interface

2-5	Design Philosophy
2-6	Basic Principles
2-6	Consistency
2-7	User Control
2-7	Naturalness
2-8	Using the Mouse
2-8	Action Paradigms
2-9	Direct Manipulation
2-9	Control Action
2-9	Target Selection
2-10	Tool Selection
2-11	Extensions
2-11	User Actions
2-11	The Keyboard
2-12	Character Keys
2-14	System Control Keys
2-14	Modifier Keys
2-15	Special Combinations
2-16	Notation
2-17	The Mouse
2-17	Clicking
2-18	Multiple-Clicking
2-19	Dragging
2-19	Moving an Object
2-19	Defining a Range
2-20	Sliding from Object to Object
2-21	Dragging from a Multiple-Click
2-21	Pressing
2-21	Modifier Keys and the Mouse
2-22	The Cursor
2-22	Mouse Scaling
2-23	Hiding the Cursor
2-23	Left and Right Orientation
2-23	Keyboard Alternatives
2-24	Standard Keyboard Alternatives
2-26	Mouse Priority

2-26	Selection
2-26	Dragging to Select
2-27	Clicking to Select
2-27	Multiple-Clicking to Select
2-28	Extending the Selection
2-28	Continuous Extension
2-29	Discontinuous Extension
2-29	Text and the Shift Key
2-30	The Interface to the Operating System
2-30	The Application Dock
2-31	The File System
2-32	Home Directories
2-32	NeXT Directories
2-33	Local and Personal Directories
2-34	Net
2-34	Paths
2-35	File Names
2-37	File Packages
2-37	The Window Interface to Applications
2-38	Window Types
2-39	Window Appearance
2-40	Window Style
2-42	Conventions
2-43	Miniwindows
2-43	Icons
2-44	Lists
2-45	Window Size
2-46	Window Ordering
2-47	Window Placement
2-47	Window Behavior
2-48	Reordering Windows
2-49	Moving Windows
2-49	Resizing Windows
2-50	Closing Windows
2-50	Miniaturizing Windows
2-51	Hiding and Retrieving Windows
2-52	Application and Window Status
2-52	The Active Application
2-53	Activating an Application
2-54	Deactivating an Application
2-54	Conditional Activation
2-55	The Key Window
2-56	The Main Window
2-58	Choosing the Key Window and Main Window
2-58	In the Active Application
2-58	When an Application Is Activated

2-59 Clicking in a Window
2-60 Working in a Window
2-60 Making a Click Unnecessary

2-60 Menus

2-61 Submenus
2-62 Keeping a Submenu Attached
2-63 Tearing off an Attached Submenu
2-63 Detaching a Submenu
2-64 Submenu Hierarchy
2-64 Commands
2-66 The Main Menu
2-66 Placement
2-67 Bringing the Main Menu to the Cursor
2-67 Standard Commands
2-69 The Window Menu
2-71 The Edit Menu
2-73 The Font Menu
2-74 The Find Menu
2-75 The Request Menu

2-75 Panels

2-76 Attention Panels
2-77 Types of Attention Panel
2-77 Attention Panel Appearance
2-78 Attention Panel Behavior
2-79 Dismissing an Attention Panel
2-79 Control Panels
2-80 Persisting Panels
2-80 Relinquishing Key Window Status
2-80 The Information Panel

2-81 Controls

2-82 Sliders
2-83 Buttons
2-86 Text Fields
2-88 Scrollers
2-89 Scroller Layout
2-90 The Knob and Bar
2-90 The Scroll Buttons
2-91 Automatic Scrolling
2-91 Fine Tuning

Chapter 2

The NeXT User Interface

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/Notes/UIUpdate/UIUpdate.rtf](#)

This chapter discusses the NeXT user interface from the programmer's point of view. It's meant to serve as a bridge between your experience as an end user of the NeXT computer and your experience writing applications for other end users.

- It explains the user interface and introduces some of its rationale so that it will be easier for you to design your application.
- It expands on terminology you're already familiar with from the *NeXT User's Reference* manual. Terminology introduced in this chapter is used without further explanation throughout the rest of the manual.
- It gives guidelines that all applications should follow. When the interface to your application is consistent with others running on the NeXT computer, users will find it more familiar, quicker to learn, and easier to use.

Much of the functionality and appearance of the NeXT user interface is built into the Window Server and Application Kit. You won't need to program the complete interface for your application; windows, buttons, scrollers, and other graphic objects are provided for you. The descriptions in this chapter will help you make the best use of these facilities.

Design Philosophy

A user interface must meet the needs of both novice and experienced users.

- For the novice or infrequent user, it must be simple and easy both to learn and to remember. In particular, it shouldn't require any relearning after an extended absence from the computer.
- For the more experienced user, it must be fast and efficient. Nothing in the user interface should get in the way or divert the user's attention from the task at hand.

The challenge is to accommodate both these goals in ways that don't conflict—to combine simplicity with efficiency.

A graphical (mouse-based) user interface is well suited to this task. Because graphical objects can be endowed with recognizable features of real objects, users can borrow on their everyday experience when they approach the computer. Graphical buttons work like you'd expect real buttons to work, windows behave much like separate tablets or sheets of paper, sliders and other graphical objects act like their physical counterparts off-screen. The computer becomes less an entirely new world with its own rules than an extension of the more familiar world away from the computer screen.

This not only makes the user interface easier to learn and remember, it also permits operations to be simpler and more straightforward. Picking an option is as easy as flicking a switch; resizing a window is as direct and simple as pulling on a tab. Thus the same attributes of the user interface that provide simplicity for novice users can also result in efficiency for more expert users.

Basic Principles

The NeXT user interface is designed with certain basic principles in mind. Four are especially important, and can be stated succinctly:

- The interface should be consistent across all applications.
- The user is in charge of the workspace and its windows.
- The interface should feel natural to the user.
- The mouse, rather than the keyboard, is the primary instrument for user interaction with the interface.

Each of these principles is spelled out in more detail in the sections that follow.

Consistency

When all applications have the same basic user interface, every application benefits. The consistency makes each application easier to learn, thus increasing the likelihood of its acceptance and use.

Just as drivers become accustomed to a set of conventions on public highways, so users tend to learn and rely on a set of conventions for their interaction with a computer. Although different applications are designed to accomplish different tasks, they all share, to one degree or another, a set of common operations—selecting, editing, scrolling, setting options, making choices from a menu, managing windows, and so on. Reliable conventions are possible only when these operations are the same for all applications.

The conventions permit users (like drivers) to develop a set of habits, and to act almost instinctively in familiar situations. Instead of being faced with special rules for each application (which would be like each town defining its own rules of the road), users can carry knowledge of how one application works on to the next application.

User Control

The workspace and the tools for working in it (the keyboard and mouse) belong to the user, not to any one application. Users should always be free to choose which application and which window they will work in, and to rearrange windows in the workspace to suit their own tastes and needs.

When working in an application, the user should be afforded the widest possible freedom of action. It's inappropriate for an application to arbitrarily restrict what the user can do; if an action makes sense, it should be allowed.

In particular, applications should avoid setting up arbitrary "modes," periods when only certain actions are permitted. Modes often make programming tasks easier, but they usurp the user's prerogative of deciding what will be done. They can thus feel annoying and unreasonable to users who aren't concerned with implementation details.

On occasion, however, modes are a reasonable approach to solving a problem. Because they let the same action have different results in different contexts, they can be used to extend functionality. When permitted, a mode should be freely chosen, provide an easy way out, and keep the user in control. On the NeXT computer, modes are used in only three situations:

- In the tool-selection paradigm, discussed under "Action Paradigms" below
- In attention panels, discussed under "Panels" later in this chapter
- In "spring-loaded" modes that last only while the user holds a key or mouse button down

Naturalness

The great advantage of a graphical user interface is that it can feel natural to the user. The screen becomes a visual metaphor for the real world; the objects it displays can be manipulated in ways that reflect the ways familiar objects in the real world are manipulated. This is what's meant when a user interface is said to be "intuitive"—it behaves as we expect it would based on our experience with real objects in the real world.

The similarity of graphical to real objects is at a fundamental rather than a superficial level. Graphical objects don't need to resemble physical objects in every detail. But they do need to behave in ways that our experience with real objects would lead us to expect.

For example, objects in the real world stay where we put them; they don't disappear and reappear again, unless someone causes them to do so. The user should expect no less from graphical objects. Similarly, although a graphical dial or switch doesn't have to duplicate all the attributes of a real dial or switch, it should be immediately recognizable by the user and should be used for the sorts of operations that real dials and switches are used for.

Each application should try to maximize the intuitiveness of its user interface. Its choice of graphical objects should be appropriate to the tasks at hand, and users should feel at home with the operations they're asked to perform. The more natural and intuitive the user interface, the more successful an application can be.

Using the Mouse

All aspects of the user interface are represented by graphical objects displayed on-screen, and all graphical objects are operated mainly by the mouse, not the keyboard. The keyboard is principally used for entering text; the mouse is the more appropriate instrument for a graphical interface.

Nevertheless, it's often a good idea to provide keyboard alternatives to mouse actions (see "Keyboard Alternatives" later in this chapter). They can be efficient shortcuts for experienced users. Keyboard alternatives are always optional, however; visual representations on the screen never are. A keyboard operation without a corresponding mouse-oriented operation on-screen isn't allowed.

One of the goals of the user interface is to extend to mouse operations the same naturalness and consistency that the keyboard provides for experienced typists. This is possible only if mouse operations follow established paradigms that users can come to rely on. The next section defines the paradigms used on the NeXT computer.

Action Paradigms

A graphical user interface works best when there are well-defined paradigms for using the mouse. The paradigms must be broad enough to encompass actions for the widest possible variety of applications, yet precise and limited enough so that users are always aware of what actions are possible and appropriate.

The NeXT user interface supports these four paradigms of mouse action:

- Direct manipulation
- Control action
- Target selection
- Tool selection

These paradigms are described below.

Direct Manipulation

Most objects respond directly to manipulation with the mouse—a button is highlighted when pressed, a window comes forward when clicked, the knob of a slider moves when dragged. Direct manipulation is the most intuitive of the action paradigms and the one best suited for modifying the position and size of graphical objects. Windows, for example, are reordered, resized, and moved only through direct manipulation.

By directly manipulating icons that represent documents, applications, mail messages, or other objects stored in the computer’s memory, users can manipulate the objects the icons represent. For example, dragging an icon to a new location can change the position of a file in the directory hierarchy.

Some objects, such as buttons and menu commands, can’t be moved or resized. They nevertheless respond to direct manipulation as a way of giving feedback to the user. The response—mainly highlighting—shows that the user’s action has successfully invoked one of the other paradigms.

Control Action

Some objects—buttons, scrollers, and text fields, among others—are vehicles for the user to give instructions to an application. By manipulating the object, the user controls what the application does. Clicking a close button, for example, not only causes the button to become highlighted, it also removes the window from the screen. The button is simply a control device—like a light switch or a steering wheel—that lets the user carry out a certain action. Graphical objects that play this role on the screen are therefore collectively known as *controls*.

The control-action paradigm is most appropriate for setting program attributes other than the position and size of graphical objects—for example, determining which font to use or whether to boot from an optical or a Winchester® disk. (See “Controls” later in this chapter for more on control objects.)

Target Selection

Some controls act on a selected domain. The user first selects what the control should act on, the *target*, then chooses the control. For example, a user might select a range of text in a file, then choose the Cut command from the Edit menu to remove it. The selection of a target always precedes the choice of a control action. Selected objects are usually editable graphics or text, but they may also be other types of objects, such as windows (the Close command) and icons (the Delete command).

Target selection is the normal paradigm for controlling or operating on objects. It has the advantage that a sequence of different control actions can apply to the same target. For example, selected text can be changed first to a different font, then to a different point size,

and then perhaps copied to the pasteboard. Moreover, a single control can act on a number of different user-selected targets, making it extremely efficient and powerful. The Cut command, for example, can delete text, as well as graphics, icons, and other objects.

Tool Selection

In this paradigm, users can change the meaning of subsequent mouse actions by selecting an appropriate tool, often displayed in a palette with several other tools. Each tool controls a certain set of operations that are enabled only after it's chosen. For example, a graphics editor might provide one tool for drawing circles and ovals, another for rectangles, and still another for simple lines. Depending on which tool is chosen, mouse actions (clicking and dragging) will produce very different visual results. The cursor assumes a different shape for each tool, so that it's apparent which one has been selected, and the tool itself remains highlighted.

The tool-selection paradigm is appropriate when a particular type of operation is likely to be repeated for some length of time (for example, drawing lines). It's not appropriate if the user would be put in the position of constantly choosing a new tool before each action.

Tool selection, in effect, sets up a mode—a period of time when the user's actions are interpreted in a special way. A mode limits the user's freedom of action to a subset of all possible actions, and for that reason is usually to be avoided. But in the tool-selection paradigm, the mode is mitigated by a number of factors:

- The mode isn't hidden; the altered shape of the cursor and highlighted state of the tool make it apparent which actions are appropriate.
- The mode isn't unexpected; it's the result of a direct user choice, not the by-product of some other action.
- The way out of the mode (usually clicking in another tool) is apparent and easy. It's available to the user at any time.
- The mode mimics the way things are done in the real world. Artists and workers choose an appropriate tool (whether it's a brush, a hammer, a pen, or a telephone) for the particular task at hand, finish the task, and choose the next tool.

Extensions

Users will come to count on a basic set of familiar operations throughout the user interface. It's each application's responsibility to make the action paradigms it uses apparent to the user—controls should look like controls (like objects that fit into the control-action paradigm), palettes of tools should be self-evident, and so on.

An application should also make certain that its paradigms fit the action. It wouldn't be appropriate, for example, to force users to choose a "moving tool" or a control action just to move an object. Graphical objects should move, as real objects do, through direct manipulation.

Properly used, the paradigms described above can accommodate a wide variety of applications. Yet over time, as programmers develop innovative software, new and unanticipated operations will require extending the user interface.

Extensions shouldn't be undertaken lightly. All possible solutions within the standard user interface described in this chapter should be exhausted first. Added functionality must be carefully weighed against the ill effects of eroding inter-application consistency for the user.

If an extension is required, it should be designed to grow naturally out of the standard user interface, and must adhere to the general principles discussed above.

User Actions

Users give instructions to the computer through their actions on the mouse and keyboard. Because these two devices are central to the user interface, they're discussed first, before windows and other graphical objects.

The Keyboard

The NeXT computer keyboard resembles a conventional typewriter keyboard, with the usual keys in their traditional locations. However, the keyboard also has many keys that aren't found on typewriters, including Command, Alternate, and Control keys, and a set of keys arranged in a calculator-style numeric keypad. Keys on the keypad are situated more conveniently for entering numbers and doing calculations than are the corresponding keys on a typewriter keyboard.

Figure 2-1 illustrates the U.S. version of the NeXT keyboard (currently the only version).

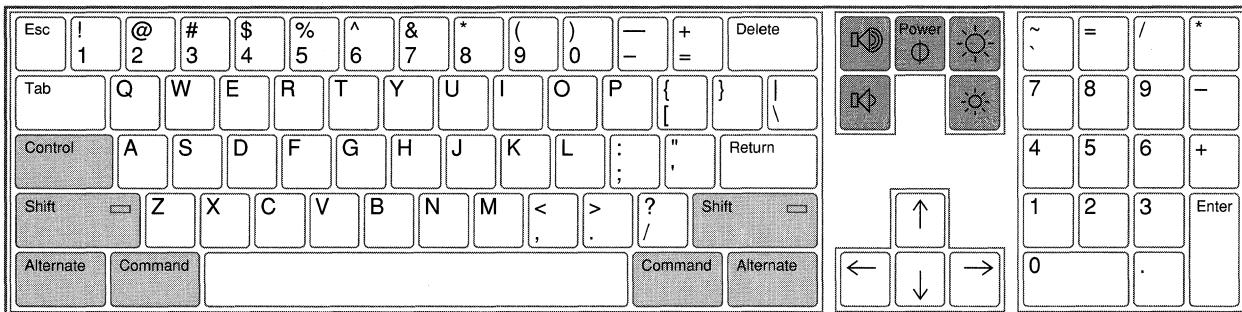


Figure 2-1. The Keyboard

As shown in Figure 2-1, there are three basic types of keys:

- *Character keys*, which transmit characters to the computer.
- *System control keys*, which control the computer's basic functions—the Power and volume keys, for example.
- *Modifier keys*, such as Shift and Command, which change the effect of a keyboard or mouse action—for example, Shift-3 yields “#”, Command-c issues a Copy command, and Alternate-click extends a selection. The modifier key is held down while another key or a mouse button is pressed.

Each of the three key types is discussed in its own section below.

The green label on the front of some keys indicates the function that the key will perform when pressed along with the Command key (which is itself labeled in green). For more information, see the section on the modifier keys.

Character Keys

The character keys generate text characters such as letters, numbers, punctuation marks, and other symbols. They include the space bar, which generates the space character (ASCII 0x20) and all the keys on the keypad. Applications recognize characters by the codes listed in the *NeXT Technical Summaries* manual. Characters generated from the keypad can be distinguished from the same characters generated on the main keyboard by an additional flag, but normally this isn't necessary.

Several special character keys, listed below, generate characters that typically perform a function—the character causes the application to make something happen. Exactly what happens depends on the application; some typical functions are mentioned here.

- The Return key generates a carriage return (ASCII 0x0D), which moves the insertion point or prompt to the beginning of the next line, much like the carriage return of a typewriter. When data is entered in a text field or form, Return informs the application that the data is ready for processing.
- The Enter key (on the numeric keypad) generates the ETX character (ASCII 0x03). Like Return, it signals that data is ready for processing. It need not move an insertion point or prompt to the beginning of the next line. (Enter can also be generated with Command-Return.)
- The Delete key generates the DEL character (ASCII 0x7F), which removes the preceding character in text or deletes the current selection. Shift-Delete generates the backspace character (ASCII 0x08), which moves the insertion point back one character. In most applications, backspace performs the same functions as Delete.
- The Tab key (and Control-I) generate the tab character (ASCII 0x09), which moves forward to the next tab stop, or to the next text field in sequence. Shift-Tab generates the back tab character (ASCII 0x19), which moves backward to the previous tab stop or text field.
- The Esc key generates the escape character (ASCII 0x1B). It's included on the keyboard for UNIX compatibility but has no direct role in the user interface. Shift-Esc generates a tilde (~).
- The arrow keys, to the right of the main keyboard, move the symbol that's used in some contexts to track where the user is writing or entering data—for example, the insertion point in a document processor. These keys generate the character codes for arrow symbols in the Symbol font (Symbol 0xAC, 0xAD, 0xAE, and 0xAF), but they should never be used to generate visible characters. When the Shift key is down, they generate the character codes for the double arrows (Symbol 0xDC, 0xDD, 0xDE, and 0xDF), but these characters also shouldn't be made visible.

Visible arrows are generated in the Symbol font by other character keys. To know what to do with an arrow character, an application must check to see which key generated the character. Since characters generated by the arrow keys are flagged in the same way as characters generated by keys on the keypad, this is fairly straightforward. See “Keyboard Event Information” in Chapter 5, “Events,” for more information.

Note: The arrow keys have nothing to do with the cursor, which is controlled only by the user's mouse movements.

If the user holds a character key down for a certain amount of time, the character is repeatedly generated in rapid-fire succession. The time the character starts to repeat and the rate at which it repeats are system-configurable; the user can set them with the Preferences application.

System Control Keys

The five system control keys are located above the arrow keys. They're illustrated in Figure 2-2.

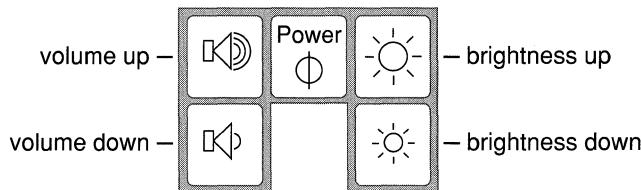


Figure 2-2. System Control Keys

- The Power key turns the computer on and off.
- The volume keys control the volume of the built-in speaker. They also control the volume at the stereo headphone jack on the MegaPixel Display. When the Command key is held down, the volume-down key toggles the built-in speaker (and only that speaker) off and on.
- The brightness keys control the brightness of the display.

All the system control keys, except Power, repeat when held down.

Modifier Keys

Modifier keys change the meaning of other keys and of the user's actions with the mouse. Unlike character keys, modifier keys by themselves don't generate characters. There are seven modifier keys: Control and two each of Shift, Alternate, and Command. Programs can distinguish between the left and right key of the Shift, Alternate, and Command pairs, but if they do, they won't be hardware-independent (for details, see "Event Flags" in Chapter 5).

To use a modifier key, the user must hold the key down and, while keeping it down, press the key (or perform the mouse action) to be modified. More than one modifier key may be used at a time—for example, Command-Alternate-3.

The modifier keys and their effects when used with other keys are presented below.

- The Shift key modifies keystrokes to produce the uppercase character on letter keys and the upper character on two-character keys. Pressing Command-Shift, and releasing the Shift key before another key is pressed, sets Alpha Lock, which in turn illuminates the green light in the Shift key. Alpha Lock turns on Shift for letter keys only. To turn Alpha Lock off, press and release Command-Shift again.

- The Alternate key modifies keystrokes to produce an alternate character to the one that appears on the key; in general, these are special characters that are used relatively infrequently. To find out which alternate characters are generated by which keys, see the *NeXT User's Reference* manual.
- The Control key modifies keystrokes to produce standard ASCII control characters. Some control characters are generated by single character keys—for example, Tab is Control-I, Return is Control-M, and backspace (Shift-Delete) is the same as Control-H.
- The Command key provides a way of choosing commands with the keyboard rather than the mouse. As an alternative to clicking a menu command with the mouse, the user can press the Command key in conjunction with the character displayed in the menu next to that command. Some standard commands are marked in green on the front of the keys which (unshifted) are the keyboard alternatives for those commands. For example, Command-c chooses the Copy command. Other standards are recommended in the “Keyboard Alternatives” section later in this chapter.

Each of the modifier keys sets a flag indicating that it was down; Alpha Lock sets a flag different from the one set by Shift. The Shift, Alternate, and Control keys usually also change the character code that the application receives (from what it would be without the modifier key). The Command key usually doesn't change the character; it simply signals that the user's intent in typing the character was to issue a command.

The Command key also disables two normal keyboard features. While it's held down:

- Keys don't repeat.
- Alpha Lock doesn't produce uppercase characters.

Special Combinations

A handful of Command-key combinations produce special effects. Some play a particular role in the user interface; others, in effect, give commands to the computer itself, rather than to just one application. The special combinations are listed below. All but the last three of these combinations are marked in green on the keyboard.

- Command-Shift sets and unsets Alpha Lock, but only when the Shift key is released before another key is pressed. (This was described in the discussion of the Shift key above.)
- Command-Return is the same as Enter.
- Command-. (period) lets users abort the current operation in some applications.
- Command-space is used for file name completion. In contexts where it's appropriate for the user to type a file name (such as in a Workspace Manager window or an Open panel), Command-space displays as many characters as match all possible file names in the directory. If the user first types enough characters to identify a particular file and

then presses the space bar with the Command key down, the remaining characters of the file name are filled in. (In many applications, the Esc key also performs file name completion.)

- Command-volume down turns the speaker off and on.
- Command-Command-` , produced by holding both Command keys down and pressing the key at the upper left of the numeric keypad, generates an NMI (nonmaskable interrupt). It brings up the NMI monitor window.
- Command-` , produced with just the Command key to the right of the space bar, displays a panel that gives the user the option of restarting the computer, turning the power off, or cancelling the command.
- Command-Alternate-* , produced by pressing the Command and Alternate keys at the lower left of the keyboard in conjunction with the * key on the keypad at the upper right, performs a reset to reboot the machine. The reset is immediate; no panel or monitor gives the user the option of cancelling the instruction.

Notation

Control characters are traditionally indicated by uppercase letters—for example, Control-I. This doesn’t mean, however, that the Shift key must be used in conjunction with the Control key. Control-I when produced with the Shift key down (or in Alpha Lock) is the same as Control-I when produced without the Shift key.

The Command and Alternate keys, on the other hand, distinguish between shifted and unshifted characters; Command-I isn’t the same as Command-i, and Alternate-I isn’t the same as Alternate-i.

Since Alpha Lock doesn’t produce uppercase characters when the Command key is pressed, it’s recommended that “Command-Shift-I” (rather than “Command-I”) be used to note the uppercase Command character in user documentation. The inclusion of “Shift” is a reminder to the user to manually press the Shift key.

In contrast, Alpha Lock works with the Alternate key. Uppercase Alternate characters therefore don’t require an explicit mention of the Shift key; documentation should use the simpler “Alternate-I” instead of “Alternate-Shift-I.”

The Mouse

The mouse controls the movement of the cursor on-screen. Typically, the user moves the cursor over an object in the workspace and presses a mouse button to make something happen. With the mouse, the user can edit documents, rearrange windows, and operate any control; the mouse is the essential tool of a graphical interface.

Users can manipulate the mouse in just two ways:

- Move it to position the cursor. The standard arrow cursor “points to” the object touched by its tip. (The cursor is also said to be positioned “over” the object at its tip.)
- Press and release the mouse buttons. The mouse that comes with the NeXT computer has two buttons, one on the right and one on the left. Initially, both buttons work alike, but they can be differentiated by the Preferences application (see “Left and Right Orientation” below).

From these two simple actions, a few basic mouse operations are derived:

- Clicking
- Multiple-clicking
- Dragging
- Pressing

Clicking

The user *clicks* an object by positioning the cursor over it, then pressing and releasing a mouse button. Usually the mouse isn’t moved during a click, and the mouse button is quickly released after it’s pressed. However, timing generally isn’t important; what’s important is where the cursor is pointing when the mouse button is pressed and released.

Clicking is used to pick an object or a location on the screen. If the object is a window, the click brings it to the front and may select it to receive characters from the keyboard. If the object is a menu command, button, or other control, the click performs the control’s action. In text, a click selects the insertion point. In a graphics editor, it may select the location for a Paste command.

When the user clicks an object on-screen, the object experiences the click as two separate user actions, one when the mouse button is pressed, and one when it’s released. The object should provide immediate graphic feedback to the user when the mouse button goes down. However, depending on the intent of the click, the object may wait for the mouse button to go back up before doing anything more:

- If the click is intended to initiate a control action or choose a tool, the object usually acts when the mouse button goes up. This gives users an opportunity to change their minds. If they move the cursor away from the object before releasing the button, the action is canceled. Suppose, for example, that a user presses the left mouse button

while the cursor points to the Cut command in the Edit menu. The command is highlighted, but nothing is cut until the mouse button is released. If the user moves the cursor outside the menu before releasing the mouse button, the command won't be carried out.

- If the click is intended to manipulate the object itself, the object reacts immediately when the mouse button goes down. For example, when a window is clicked, it comes to the front of the screen without waiting for the mouse button to go up. Similarly, when editing text, the user is committed to a new selection as soon as the mouse button is pressed.

Multiple-Clicking

The user *double-clicks* an object by positioning the cursor over it, then quickly pressing and releasing a mouse button twice in succession. The mouse button must be pressed the second time within a short interval of the first, or the action will count as two successive clicks rather than a double-click. In addition, the cursor can't move significantly during the interval; this is to guarantee that the double-click remains focused on a single location on-screen.

With the Preferences application, users can set the maximum length of the time interval to suit their individual needs.

The user *triple-clicks* an object by rapidly pressing and releasing a mouse button three times in succession. The time interval between successive clicks and the distance the cursor can move between the first and the last click are subject to the same constraints that apply to a double-click.

Double-clicking should be used only for actions that logically extend the action of a single click, and triple-clicking only for actions that extend a double-click. There are two reasons for this rule, one philosophical, the other programmatic:

- Complex mouse actions are best remembered and understood when they appear to grow naturally out of simpler actions.
- Every double-click includes a single click (the first click in the sequence), and every triple-click includes a double-click. At the time an application receives one click, it can't know that any others are on their way. So it must first act on the single click, then the double-click, then the triple-click.

For example, double-clicking an icon in a Workspace Manager window picks out that icon just as a single click would. It then goes on to open the application associated with the icon. A single click in text selects an insertion point, a double-click extends the selection to a word, and a triple-click extends it further to a full line, sentence, or paragraph.

Quadruple clicks (and above) become increasingly difficult for users to produce or understand. They're neither used nor recommended in the NeXT user interface. Triple-clicks should be used only sparingly.

Dragging

The user *drags* by pressing a mouse button and moving the mouse (and cursor) while the button is down. Dragging is used in a variety of situations, principally these three:

- To move an object, such as a window or the knob of a scroller
- To define a range, usually to select the objects falling within the range
- To slide from one object to another, in order to extend an action initiated in the first object to the second object

Moving an Object

The user can drag an object by positioning the cursor over it, pressing the mouse button, and moving the mouse while the button is down. The object moves so that it remains aligned with the cursor on-screen. If the object is constrained within a particular area or track—as is a scroller knob, for example—it remains as closely aligned with the cursor as possible.

Every dragging action implies a click; the mouse button goes down to initiate dragging and back up again to end it. If an object responds to both clicking and dragging, every time it's dragged it will also respond to the click. Dragging a window, for example, also brings it to the front.

Defining a Range

The user can also drag over an area or through a series of items (such as text characters) to define a range. The action here is the same as for dragging an object: The mouse is moved while the mouse button is held down. The position of the cursor when the mouse button is pressed is the *anchor point*; its position when the mouse button is released is the *endpoint*. The difference between the anchor point and endpoint determines the area or objects inside the range.

Applications often drag out—or “rubberband”—a rectangle to show the area covered between the anchor point and endpoint. This is illustrated in Figure 2-3.

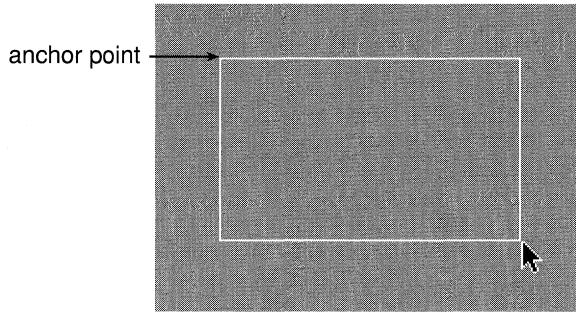


Figure 2-3. Rubberbanding

Dragging to define a range is mostly used to make a selection (such as a string of text characters or a group of icons) for the target-selection paradigm.

Sliding from Object to Object

Usually, for a click to choose an object in the control-action or tool-selection paradigm, the mouse button must be both pressed and released as the cursor points to the object. In some cases, however, users can press the mouse button as the cursor points to one object, then drag to another object before releasing the button. Dragging serves to extend the action over two or more related objects. The object under the cursor when the mouse button goes up is the one that responds as if it were clicked.

For example, a user can choose a menu command by pressing the mouse button as the cursor points to one command and releasing it as it points to another. Users can similarly drag from one tool to another tool when they're displayed together in a palette.

Controls that are presented to the user in a group can act in the same way. For example, a user can drag from object to object in a set of mutually exclusive buttons or switches. Dragging through a group of sliders arranged like a graphic equalizer might reset all of them.

Sometimes it's possible to drag from one type of object into another. A user can drag from a button that controls a pop-up list through the list to make a selection, or from a menu command that controls a submenu into the submenu.

In each case, the object the cursor points to when the dragging action begins and the object it points to when the dragging action ends are part of a single functional entity. The manner in which the objects are displayed should make this unity apparent.

Dragging from a Multiple-Click

The act of pressing a mouse button to initiate dragging can be part (the last part) of a double-click or triple-click. If the user doesn't immediately release the mouse button and begins dragging at the end of a multiple-click, the dragging action can be assigned a meaning that's related to the meaning of the multiple-click.

For example, double-clicking selects a word in editable text, and dragging from a double-click selects additional words within a range of text. If triple-clicking selects a line, dragging from a triple-click will select additional lines within the range.

Pressing

The user *presses* an object on-screen by positioning the cursor over it, pressing a mouse button, and holding the button down for a period of time. Although pressing implies clicking (since the mouse button must be released sometime), an object is said to be pressed rather than clicked if releasing the mouse button too soon would cut the action short. Control objects that respond to pressing act immediately when the mouse button is pressed; they don't wait for the button to go up.

For the most part, pressing is an alternative to repeated clicking. It should be used wherever a control action can be repeated with incremental effect. For example, clicking a scroll button scrolls one line of text, clicking the button again scrolls another line, and so on; pressing the scroll button scrolls lines repeatedly for a continuous action until the mouse button is released.

Pressing is also used to initiate the action of sliding from one object to another. If a button controls a pop-up list, the user presses the button and drags through the list to choose one of its options. After pressing a menu command to attach a submenu, the user can drag into the submenu.

Modifier Keys and the Mouse

Applications can detect when the user is holding down a modifier key while clicking, dragging, or pressing, and can choose to assign a different meaning to the modified mouse action. Modified mouse actions implement only optional or advanced features of the user interface (because they're harder to remember and require more coordination to produce). They typically extend or alter the effect of the unmodified mouse action. For example:

- Dragging a file icon from one directory window to another moves the file to the new directory; Command-dragging copies the file instead.
- Clicking a scroll button scrolls a line of text; Alternate-clicking scrolls a larger amount.

- Dragging a window by its title bar moves the window, brings it to the front, and activates its application; Alternate-dragging moves the window and brings it to the front, but doesn't activate its application.
- Clicking selects a new insertion point in text; Alternate-clicking extends the selection to include everything between the current insertion point and the point of the click.
- Clicking selects an icon in a directory window; Shift-clicking adds new icons to the current selection.

Alpha Lock doesn't work for Shift-clicking (or Shift-dragging); the Shift key must be held down manually. This way, users won't find themselves Shift-clicking by mistake when they intend only to click.

Although applications can use the Control key to modify a mouse action, the other modifier keys are preferred for this wherever possible.

The Cursor

An application can change the cursor from the standard arrow to any other image of an equal size (16 pixels by 16 pixels on the MegaPixel Display). When doing so, it must specify what point in the cursor acts like the tip of the arrow. That point, the cursor's *hot spot*, should be apparent to the user from the shape of the image. For example, if the cursor is an "X", the hot spot would be where the two lines cross.

For some types of applications, a shape other than an arrow might be more convenient. For example, an I-beam cursor is handier for positioning between characters. Its hot spot is in the center of the beam.

It's often a good idea to change the shape of the cursor to indicate that the user has entered a mode. In applications that use the tool-selection paradigm, the cursor should change to indicate which tool has been selected. For example, the cursor might look like a pencil while thin lines are being drawn in a graphics application, or like a wide brush when painting in broad strokes.

If mouse actions are valid only in a certain area, the cursor should revert to its normal shape when it leaves the area. It's best not to change the cursor too often, however. To avoid confusing the user, stick with the standard arrow wherever reasonable.

Mouse Scaling

The cursor moves on-screen when the user moves the mouse; but the ratio of the two movements isn't one-to-one. Rapid mouse movements move the cursor farther than slow ones. Users can set the *mouse scaling*, how responsive the cursor is to mouse movements at different speeds, with the Preferences application.

Hiding the Cursor

A visible cursor is essential for mouse actions, but it can get in the way when the user is concentrating on using the keyboard. Therefore, applications that let users enter or replace text normally hide the cursor—make it disappear from the screen—when the user begins typing. A hidden cursor returns to the screen as soon as the user moves the mouse, signaling a shift in attention away from the keyboard back to the mouse.

The cursor is also hidden whenever the user selects an insertion point or a range of text. A new selection is a good indication that the user is ready to begin typing again. Hiding the cursor when the user selects a new insertion point avoids confusion between the I-beam cursor and the vertical bar representing the insertion point. Unless it's hidden, the I-beam can obscure the vertical bar.

Left and Right Orientation

To start, the two buttons of the mouse work identically; either button can be used for the ordinary operations of clicking, dragging, and pressing.

The two buttons can be differentiated with the Preferences application. Users can enable one of the buttons, either the right or the left, for the special function of bringing the main menu to the cursor (see “The Main Menu” later in this chapter for details). Thereafter, the enabled button has only that function; it can't be used for ordinary mouse operations. This leaves the other button as the one that will be primarily used.

Users generally feel most comfortable operating the mouse with the index finger, and therefore prefer to keep the button nearest that finger as the primary button. For right-handed users, the left mouse button is nearest the index finger; for left-handed users, it's the right mouse button. Therefore, right-handed users generally enable the right mouse button to bring the main menu to the cursor, and left-handed users enable the left mouse button.

In general, documentation takes the point of view of a right-handed user. The primary mouse button is referred to as the “left” mouse button and the events it generates as “left mouse events.”

Keyboard Alternatives

A graphical user interface is easy for most people to learn and remember. Objects have a familiar look on the screen and behave in a way that's reminiscent of the real-world objects they emulate. However, many users find it faster and easier to operate graphical objects using the keyboard rather than the mouse. For this reason, it's often appropriate to provide keyboard alternatives to the mouse, at least for common operations.

Keyboard alternatives consist of a single keystroke, modified by the Command key (and possibly another modifier key). The Command key is required so that keystrokes that make something happen (give commands) are clearly separated from those that enter data (cause typing to appear).

Any character can be used in combination with the Command key. If the character is a letter, it can be either uppercase or lowercase, although lowercase characters are preferred because they don't require the user to press two modifier keys (Shift and Command) at once.

Note: Alpha Lock doesn't affect the character typed with the Command key. A manual Shift is required to produce an uppercase keyboard alternative. Lowercase keyboard alternatives are generated even while Alpha Lock is on.

Keyboard alternatives are allowed only for the commands in a menu, the buttons in a panel, or the items in a pull-down list. (Menus, panels, and pull-down lists are described under "Window Types" later in this chapter.) The characters used as a keyboard alternatives must be displayed to the user in the menu, panel, or list. Menus put them on the commands themselves and pull-down lists follow this example. A panel can present the keyboard alternatives for its buttons in any way that's appropriate to the design of the panel.

Standard Keyboard Alternatives

NeXT has reserved two groups of keyboard alternatives for some common commands. The first group is listed in the table below along with the commands they perform and the menus where the commands are located. (See "Menus" later in this chapter for more information on the listed commands and menus.)

Keyboard Alternative	Command	Menu
Command-a	Select All	Edit menu
Command-b	Bold (Unbold)	Font menu
Command-c	Copy	Edit menu
Command-h	Hide	main menu
Command-i	Italic (Unitalic)	Font menu
Command-o	Open	Window menu
Command-p	Print	main menu
Command-s	Save	Window menu
Command-t	Font Panel	Font menu
Command-v	Paste	Edit menu
Command-w	Close	Window menu
Command-x	Cut	Edit menu
Command-z	Undo	Edit menu

The keyboard alternatives shown above must be used for the listed commands, and can be used only for those commands. If your application implements the functionality that a command represents, it must provide both the command and the keyboard alternative.

This means, for example, that if your application has windows with close buttons (as almost all applications do), it must have a Window menu with a Close command and Command-w as the command's keyboard alternative. If your application opens files, it must have an Open command with Command-o as the keyboard alternative. If it doesn't allow the user to open files, it won't have an Open command and must forgo using Command-o as a keyboard alternative. Command-o is associated only with the Open command.

Keyboard alternatives in the second group are less severely restricted. They're listed in the table below:

Keyboard Alternative	Command	Menu
Command-=	Define in Webster	Request menu
Command-d	Find Previous	Find menu
Command-e	Enter Selection	Find menu
Command-f	Find Panel	Find menu
Command-g	Find Next	Find menu

If an application implements the functionality that one of these items represents, it must make use of the listed keyboard alternative. For example, if your application has a Find panel, you must provide Command-f as a way of bringing the panel up.

However, if an application doesn't implement the particular functionality of an item (if it doesn't have a Find panel, for example), it can use the keyboard alternative (Command-f) for something else. Nevertheless, to preserve inter-application consistency, it's strongly recommended that you first try to use characters that don't overlap with those on this list.

Note: In addition to the listed keyboard alternatives, many applications use Command-n for New in the Window menu, Command-q for Quit in the main menu, and Command-? for Help, also in the main menu.

Most of the keyboard alternatives listed in the two groups above are formed from the first letter of the commands they perform. Those that use another character are mnemonic in another way, or are conveniently situated near a related keyboard alternative on the keyboard. For example:

- The letter “x” is used to cross out or delete material, making it an appropriate keyboard alternative for the Cut command. It's located near “c” (for the related Copy command) on the keyboard. The letter “v” (for Paste) is also located near “c”; some users remember it as an upside-down caret.
- The letter “z”, the last letter in the alphabet, is used to undo the last series of editing changes (since the user last changed the selection).
- The letter “f” is used to bring up the Find panel. The key to its right on the keyboard, “g,” is used for the Find Next command, and the key to its left, “d,” is for Find Previous.
- The letter “a” stands for the “all” of Select All, and some users associate the letter “t,” which brings up the Font panel, with “type” or “typeface,” or with the last letter of “font.”

Mouse Priority

The user interface is visual, so all operations—all menu commands and scrolling operations, for example—have a graphical representation on-screen and can be performed using the mouse. Keyboard alternatives are just that: alternatives. They should never be used for operations that can't be performed using the mouse.

A keyboard alternative must accomplish exactly the same thing as the mouse. Even slight variations between a mouse action and its keyboard alternative run counter to the principle that every keyboard operation must match a corresponding mouse operation.

Although keyboard alternatives are tied to a graphic representation, they don't require the representation to be on-screen. Keyboard alternatives for menu commands and panel buttons work even if the menu or panel is hidden.

Selection

Users select graphical objects by clicking and dragging with the mouse. A variety of objects can be selected, including:

- Windows
- Tools in a palette
- Cells in a matrix or fields in a form
- Icons in a directory window
- Items in a list (of files or mail messages, for example)
- Characters in editable text
- Graphical elements of editable artwork

Selecting an object simply picks it out and distinguishes it from others of the same type; it doesn't change the object in any way. Most selections pick out targets for subsequent actions in the target-selection paradigm.

If users are allowed to insert new material into a display, they can select not only objects already displayed, but also locations for the insertions. For example, it's possible to select characters that have been typed into a text field, or the point where new typing should appear.

This section concentrates on how selections are made in editable material, but the rules often carry over to other types of selection as well.

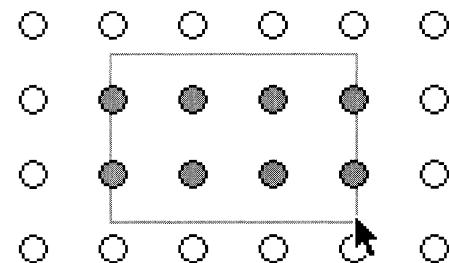
Dragging to Select

Dragging selects everything in the range defined by the anchor point (where the cursor was located when the mouse button was pressed) and the end point (the cursor's location when the mouse button is released).

What “everything in the range” means depends on the type of material selected. In serially arranged material—such as characters in connected text—the selection includes the entire series between the anchor and end points. For material consisting of objects that can be independently arranged—such as icons or the graphic elements that make up a picture—the selection includes everything within a rectangle defined by the anchor and end points. The highlighted material in Figure 2-4 shows the difference between selection in text and graphics.

ras when warm, wet climatic co
s died, they accumulated on the
ne cycles of decay, new growth,
al to build up. Enormous press
ial, and gradually many of the a
duced. In time the layers were
carbon that was present in the

Selecting text



Selecting graphics

Figure 2-4. Dragging to Select Text and Graphics

Clicking to Select

If the anchor point and end point are substantially the same—as they are for a click—the user’s action may sometimes select the item under the cursor and sometimes simply select that location. In a graphics editor, for example, a click can select an existing figure or a location to insert a new one.

In text, a click always selects a location—an *insertion point*—where characters can be entered from the keyboard. The insertion point is normally marked by a blinking vertical bar located between characters. If the user clicks on top of a character, the insertion point is adjusted to the nearest character boundary. Clicking in a margin, or in an empty area away from any text, puts the insertion point next to the nearest character in series.

Multiple-Clicking to Select

Although a single click selects only an insertion point in text, multiple-clicking selects characters already inserted. A multiple-click always selects a linguistically meaningful unit. Normally, double-clicking selects a word, and triple-clicking selects a paragraph (all the text between Return characters).

If the user drags from a multiple-click, additional units of the same type are selected. For example, double-clicking a word selects the word; dragging from the double-click then selects every other word that’s even partially within the range defined by the anchor and end points.

Extending the Selection

Normally, as soon as users commit themselves to a new selection by pressing a mouse button (to begin clicking or dragging), the current selection is canceled in favor of the new one. However, when the Alternate or Shift key is held down, the current selection is extended, not canceled.

Continuous Extension

Clicking and dragging with the Alternate key down results in a new selection that's a continuation of the previous one. The new selection includes the previous selection and everything lying between it and the location of the cursor when the user releases the mouse button. The Alternate key is thus an alternative to dragging as a way of selecting a range—the user can click to establish an anchor point, hold down the Alternate key, and click again to determine the end point.

If the previous selection is already a range, Alternate-clicking and Alternate-dragging move the edge of selection that's closest to the cursor when the mouse button goes down to the cursor's location when the mouse button goes up. The Alternate key thus also provides a way of adjusting the boundaries of the previous selection. Alternate-clicking outside a selected range extends the range to the point of the click. Alternate-clicking inside a selected range repositions the closest edge of the selection to the point of the click.

Alternate-clicking is illustrated in Figure 2-5.

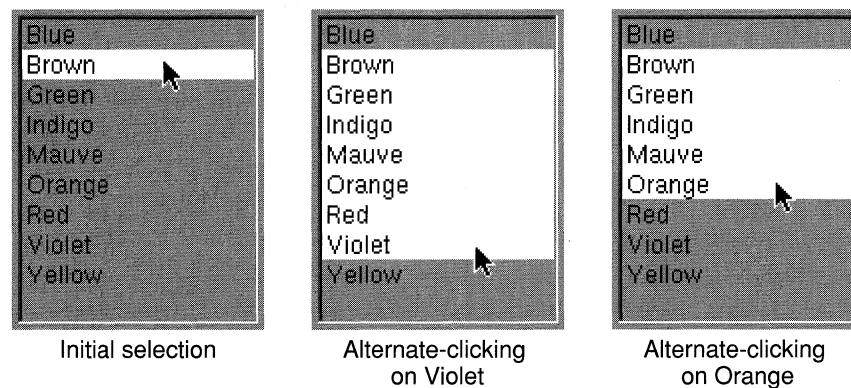


Figure 2-5. Extension with the Alternate Key

If the current selection is the result of a multiple-click, the Alternate key extends it just as dragging would. Double-clicking a word, holding the Alternate key down, and clicking another word elsewhere in the text extends the selection to include both words and all those between.

Discontinuous Extension

The Shift key lets users add to, or subtract from, the current selection. Additions don't have to be continuations of the current selection, so discontinuous selections can result.

To add to the selection, the user clicks and drags as usual while holding the Shift key down. New material is selected, but the previous selection also remains. This is illustrated in the middle column of Figure 2-6.

To subtract from the selection, the user holds the Shift key down while clicking or dragging over the current selection. Shift-clicking and Shift-dragging deselect material that's already been selected. While keeping the Shift key down, the user can first select material, then deselect it, then select it again.

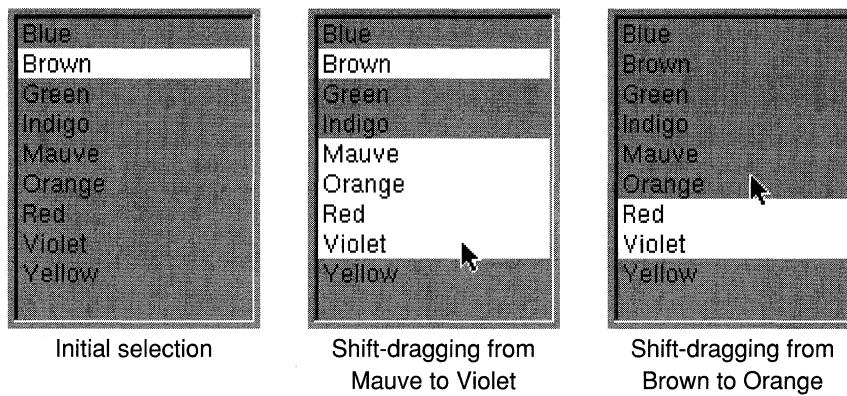


Figure 2-6. Extension with the Shift Key

Shift-dragging either selects or deselects; it never does both. Which it does depends on the item under the cursor when the mouse button goes down (the item at the anchor point):

- If the item isn't currently part of the selection, Shift-dragging serves to select it and everything the user drags over. It won't deselect material that happens already to be selected.
- If the item is currently selected, Shift-dragging deselects it and any other selected material that's dragged over. It won't add unselected material to the selection.

Text and the Shift Key

Many applications support only continuous selection for text. Discontinuous selection is more common for editable graphics, icons, and items arranged in a list.

As a convenience for users, if an application doesn't support discontinuous selection when the user is editing text, it should make the Shift key, in addition to the Alternate key, available for continuous selection.

The Interface to the Operating System

One of the goals of the NeXT user interface is to provide a simple, graphical interface to the UNIX operating system. This is largely the responsibility of Workspace Manager, the NeXT application that's brought to the screen after the user logs in. More specifically, Workspace Manager provides a graphical interface to the file system. It's a substitute for a UNIX command interpreter (or "shell") that locates files, displays directory contents, associates files with applications and icons, launches applications, and keeps track of the user's home directory and working environment. It lets users manage files by manipulating their iconic representations.

For command-line interaction with the operating system, users can put up a window that emulates a VT-100™ terminal and runs a standard shell. This window is provided by the Terminal application, which can be launched from Workspace Manager.

Terminal is documented in the *NeXT Development Tools* manual. Workspace Manager is documented in *User's Reference*.

The Application Dock

Workspace Manager maintains an *application dock*, a strip where the user can keep icons for commonly used applications, along the right edge of the screen. The dock has slots for 12 icons under the NeXT logo at the top. The logo serves both as the icon for Workspace Manager and as a handle where users can grab the dock to drag it up and down in its track.

The dock stays in front of most windows, so the icons in it are readily available; it's a convenient place for users to put the tools they favor for working on the computer. By dragging icons for selected applications into the dock from Workspace Manager's directory windows, users can customize it for their own needs. When a user logs out, Workspace Manager remembers what icons were in the dock and restores them when the user logs in again.

To launch an application, an icon in the dock works just like an icon in a directory window; double-clicking it starts up the application. But once the application is running, the docked icon assumes other functions. Mainly, it provides a way of quickly retrieving hidden or obscured windows belonging to the application. (See "Hiding and Retrieving Windows" later in this chapter for details.)

Because the icon has additional responsibilities for running applications, Workspace Manager places a freestanding icon on the screen when it launches an application, if there isn't an icon for it already in the dock. If more than one copy of an application is running, each process has its own icon.

The freestanding icon can be moved into the dock just like an icon from a directory window. If it isn't moved to the dock, it will disappear when the application terminates.

Docked and freestanding icons are illustrated later in Figure 2-13.

The File System

The file system on the NeXT computer is arranged very much like a traditional UNIX file system (see the *Technical Summaries* manual for details on how they differ).

However, by default, when users view the file system in Workspace Manager, they won't see many of the traditional UNIX directories. Instead, they'll see only a small number of directories that organize the tools and information they most need. Under the root (/) directory, they'll find the directories shown on the left in Figure 2-7:

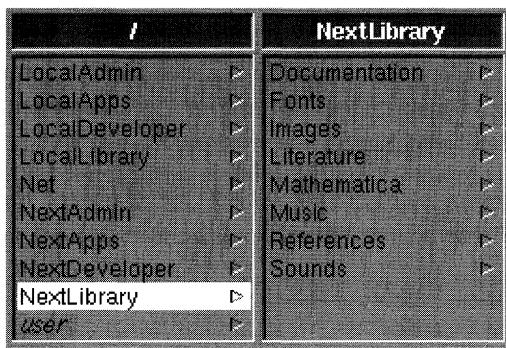


Figure 2-7. Top-Level Directories

Although users can choose to see the entire UNIX directory structure in Workspace Manager, and have an unrestricted view of it in a Terminal window, these few directories are all that most users will need. But when you begin to program for the NeXT computer, you may want a less restricted view of the file system. Header files, libraries, compilers, debuggers, and other programming tools are located in their traditional UNIX directories. To see these directories in a Workspace Manager window, set the UNIX Expert switch in the Preferences application and log in again.

Home Directories

In Figure 2-7, the directory labeled “*user*” stands for the user’s home directory. The home directories of other users may also be visible. As is traditional, home directories bear the name of the user—the name that the user logs in under.

If a machine is connected to a file server over a network, users’ home directories would more likely be located somewhere on the file server. Figure 2-7 simply illustrates the default location for a home directory and where it would be located if it were on the startup disk. Home directories on remote machines are accessed through the **Net** directory described below.

NeXT Directories

Four of the top-level directories begin with a “Next” prefix. They contain documents, resources, and applications that are bundled with the computer. The **NextApps** directory contains supported applications for NeXT users. In it you’ll find general-interest applications like Digital Webster™, *Mathematica*®, WriteNow®, Preferences, and PrintManager, as well as applications used for programming, such as Interface Builder and Edit. The programming applications in **NextApps** are ones that even users who don’t consider themselves to be “programmers” might be interested in.

The **NextAdmin** directory also contains supported applications, but ones that will be used mainly by system administrators and network managers. They’re not for general users like those in **NextApps**.

NextDeveloper has three subdirectories:

Apps	Holds applications that will be used solely by programmers. Debuggers, profiling tools, language-specific editors, and other window-based programming tools belong here. Like all other applications in the four “Next” directories, these applications can be run from Workspace Manager; command-line debuggers and profiling tools belong in standard UNIX directories such as /bin and /usr/bin .
Demos	Contains programs that demonstrate the capabilities of the NeXT computer. These aren’t full applications and aren’t supported by NeXT. However, they include games and utilities that users, not just developers, may find interesting.
Examples	Contains example programs for the NeXT computer. Subdirectories contain source code that you can study and compile.

The **NextLibrary** directory contains resource files organized into several subdirectories. The subdirectories are shown in the second column of Figure 2-7 and are described below:

Documentation	Holds technical documentation for the NeXT computer. You'll find the NeXTstep reference manual on-line in the Documentation/NextDev/NextStep directory.
Fonts	Holds the PostScript fonts available to all applications.
Images	Has files containing graphic images, in both Encapsulated PostScript format (EPS) and Tag Image File Format (TIFF).
Literature	Stores books that can be searched by the Digital Librarian. The complete works of Shakespeare are here.
<i>Mathematica</i>	Contains packages and sample notebooks for the <i>Mathematica</i> application.
Music	Holds musical pieces written in the ScoreFile language and musical performances stored as Musical Instrument Digital Interface (MIDI) data.
References	Stores reference material, including the dictionary and thesaurus for the Digital Webster application.
Sounds	Holds sound files available to all applications, including the file for the system "beep."

The four "Next" directories are reserved by NeXT. If you store your own files there, they'll be overwritten in future system updates.

Local and Personal Directories

The four directories with a "Next" prefix can be matched by four identical directories with a "Local" prefix. Internally, these four directories are organized like their "Next" counterparts. But instead of containing files supplied by NeXT, they hold information and applications provided for all users at a local site. Any user who logs in to a machine, or boots from it over a network, has access to its "Local" directories. If you add a new font for all users of your network, for example, it would reside in **/LocalLibrary/Fonts**.

"Local" directories are created as they're needed. They're not included on the release disk.

Users can add unprefixed **Library** and **Apps** subdirectories in their home directories to hold information and applications that they alone have access to. Users who develop utilities for their own use or purchase private copies of a word processor and spreadsheet, for example, should put them in **~/Apps**.

Net

The **Net** directory gives the user access to file systems that are physically located on remote machines. The immediate subdirectories under **Net** name the machines where the file systems are located. The next level of subdirectories name the root directories of the file systems on those machines. For example, **/Net/willow/misc** is where the **misc** file system located on the **willow** computer would be mounted.

Net has subdirectories only if the user's computer is set up to be connected to other machines over a network.

All the directories illustrated in Figure 2-7 above, including **Net**, are physically located on the disk that the user's machine was booted from. If a user boots from a local optical disk, for example, **NextLibrary** and all its subdirectories will be stored on the optical disk. Remote directories are mounted only under **Net**.

Paths

On UNIX, an environmental variable, **path**, holds an ordered set of directories that are searched for executable files. The default path used by Workspace Manager lists these six directories:

```
~/Apps  
/LocalApps  
/NextApps  
/NextDeveloper/Apps  
/NextAdmin  
/NextDeveloper/Demos
```

Workspace Manager uses this path in two ways:

- To find the icons it should display for files associated with a particular application. Icon information is embedded within a segment of the application executable.
- To find the application it should launch when the user double-clicks on a file.

Before using the search path to find which executable to launch, Workspace Manager first looks in the dock. Each icon in the dock represents a particular application residing in a particular directory on disk. By putting an icon in the dock, the user has indicated a preference for that version of the application over any others.

If an application isn't in the dock, Workspace Manager looks next in the current working directory, the directory containing the file the user wants to open. Only after failing to find the application there does it turn to the path listed above.

In the path, Workspace Manager looks first in the **Apps** subdirectory of the current user's home directory. That's where the user's own applications would be. It next looks in the **LocalApps** directory for site-wide software, then in the **NextApps**, **NextDeveloper/Apps**, **NextAdmin**, and **NextDeveloper/Demos** directories for NeXT-supplied software. This ordering of directories lets users customize their software to override site-wide software, and lets site-wide software override software supplied by NeXT.

Similarly ordered paths are used in other contexts. If an application requires a particular sound file, for example, it should search for it first in **~/Library/Sounds**, then in **/LocalLibrary/Sounds**, and only then in **/NextLibrary/Sounds**.

Users can alter the path shown above for Workspace Manager by setting a value for the ApplicationPaths parameter in their defaults database. You might do this, for example, to add a **/LocalAdmin** or **/LocalDeveloper/Apps** directory to the path. See Chapter 10 for more on the defaults database.

File Names

On UNIX (and other operating systems), particular types of files are sometimes identified by a characteristic file name extension. The extension is the last period in the file name and all the characters that follow it. Files with the same extension are presumed to be of the same type.

Workspace Manager uses file name extensions not only to identify particular types of files, but also to associate document files with applications. Every application that defines its own data format should append an identifying extension to the names of its document files.

These standard extensions have a well-established history:

.a	A C library file
.asm	A file containing source code for the digital signal processor (DSP)
.c	A file containing C source code
.f or .F	A file containing Fortran source code
.h	A C header file
.lnk	A file with relocatable object code for the DSP
.lod	A program that can be loaded to run on the DSP
.lst	A file with the listing for a DSP program
.m	A file containing Objective-C source code
.o	A file containing binary, relocatable object code
.p	A file containing Pascal source code
.r	A file with source code for the Ratfor variety of Fortran
.s	A file containing assembly source code for the main processor

To these NeXT adds a variety of its own extensions, including these common ones:

.app	A directory containing an executable file
.dsp	A file with binary code for the DSP
.eps	A file containing Encapsulated PostScript code
.ma	A text file for a Mathematica notebook
.mb	A file containing binary information for a Mathematica notebook
.mbox	A file containing mail messages
.midi	A file with binary MIDI data
.nib	An archive file produced by Interface Builder
.ps	A file containing PostScript code
.psw	A file containing declarations for the pswrap utility
.pswm	A file with pswrap declarations and Objective-C source code
.rtf	A file in Rich Text Format (RTF)
.score	A music file in the ScoreFile language
.snd	A sound file
.tiff	A file in Tag Image File Format (TIFF)
.wn	A file in WriteNow format
.wndict	A WriteNow dictionary

This list is open-ended. Other extensions will be added in the future.

Your application should use its own unique file name extensions to identify (and help Workspace Manager identify) its documents. To request that an extension be registered and reserved for your use, write to:

NeXT Technical Services
Extension Registry
900 Chesapeake Drive
Redwood City, CA 94063

You can also send your request by electronic mail to **ask_next@NeXT.COM** or
...!next!ask_next.

The request should include the following information:

- The file name extension you want to register. List a first and a second choice.
- The name of the application the request is for.
- Your name and the name of your company.
- Your postal address and your electronic mail address, if you have one.
- Your telephone number.

You'll be informed when the extension is registered, or if it can't be for any reason. NeXT Technical Services will make the list of registered extensions available to you so that you can choose an extension not already assigned.

Registering a file name extension reserves it for your use. If you fail to register an extension that you intend to use, someone else may register and use it instead.

File Packages

A *file package* is a directory that Workspace Manager presents to users as if it were a file. Because users normally don't look inside a file package (unless they explicitly open it as a directory), they're not likely to alter or reorganize its contents.

An application should create a file package when it has a group of files that it needs to keep together. For example, if your application displays help information that's stored in independent text files, or if it makes use of a private utility program, or if it just loads archived objects from Interface Builder ".nib" files, you may want to keep these auxiliary files in close proximity to the application executable. A file package is the way to do it.

Similarly, if your application creates documents that are split into more than one file—for example, if text is in one file and artwork in another—these files can also be grouped in a file package.

A file package for an application executable should have the same name as the executable file, plus a ".app" extension. When the user double-clicks a package with this extension, Workspace Manager looks inside it to find the executable. Since a package might contain more than one executable file, Workspace Manager recognizes the one to launch from the name of the package.

File packages for documents should bear the same extension that's assigned to the application's document files. For example, WriteNow's file packages have a ".wn" extension. Workspace Manager doesn't require there to be an identically named file within the package, but an application can impose this, or any other, requirement on its own packages. Opening and naming files within a document file package is entirely the application's responsibility.

The Window Interface to Applications

The NeXT user interface is window-based. Applications present themselves visually to users by drawing in page-like rectangles that can be moved around the screen and stacked on top of each other much like separate tablets or sheets of paper. Each rectangle, or *window*, is placed on the screen by a particular application, and each application will most likely own a variety of different windows. They are, in a sense, windows into the application.

This section introduces windows in the user interface. Later sections concentrate on particular kinds of windows, on the objects displayed within windows, and on how the windows of one application or another can be selected by the user.

Window Types

There are several different types of windows, distinguished by their appearance, by their behavior in response to user actions, and by the roles they're assigned within an application.

The ordinary windows where the main work of an application is done are known as *standard windows*. They're what users normally think of when "windows" (without any modifier) are referred to. Standard windows are the most widely used type of window and the principal type for all applications. If an application lets the user edit files, each file will be displayed in a separate standard window. If the application is a game, the game board will be in a standard window, and if the application is a simple accessory like a clock, the clock face will occupy a small standard window of its own.

Most applications will also have other windows that support the work done in standard windows. These supporting windows are summarized in the list below and are illustrated later in Figures 2-9 through 2-15.

<i>Panel</i>	A window that's used to give instructions to the application—for example, the Font and Find/Replace panels of WriteNow—or that presents information to the user—for example, the Info and Help panels of many applications.
<i>Menu</i>	A window that contains a list of commands. Menus are implemented as a special type of panel and play a special role. They give the user access to all parts of the application.
<i>Pop-up list</i>	A menu-like list of options that appears on top of a button when the button is pressed and disappears when the button is released. While the mouse button is down, the user can drag through the list to make a selection. The button displays the last item selected, and the list pops up so that the item is over the button.
<i>Pull-down list</i>	A menu-like list that appears under a button when the button is pressed. The user can drag down into the list to choose an action. The button displays a constant title for the items in the list, much like the title bar of a menu.
<i>Miniwindow</i>	A small, titled picture that represents a window that's been miniaturized. Miniaturizing a window removes it from the screen and replaces it with its miniwindow. Double-clicking the miniwindow returns the unminiaturized window to the screen.
<i>Freestanding or docked icon</i>	An application icon that Workspace Manager has placed on the screen for a running application, or one that has been dragged from a directory window (or from its place on the screen) into the dock. Freestanding and docked icons represent whole applications much as miniwindows represent single windows. All freestanding and docked icons, like the dock itself, belong to Workspace Manager.

Panels and menus each have a special subtype that's distinguished from other members of the group:

<i>Attention panel</i>	A panel that requires some user action before work can be done in any other window of the application—for example, the window that asks whether the user wants to save changes to a document before quitting. When the user acts, the panel disappears.
<i>Main menu</i>	The principal menu of the application, the one that bears the name of the application (or an abbreviation) in its title bar. It's the one menu that's guaranteed to be on-screen while the user is working in the application. Through a system of submenus, the main menu gives the user access to all of the application's other menus and most of its panels.

In addition to these window types, users will also see the *workspace window*, the normally dark gray background to all the other windows on-screen.

In documentation for users, the term “window” generally refers only to standard windows, though panels and menus are acknowledged to be windows of a special type. Miniwindows, lists, and icons are referred to only by their specific names; they should not be included within the generic term “window” as this would imply common behavior that's lacking.

Window Appearance

Every window has a *content area*, where the application is free to draw (although the Application Kit draws default miniwindows and icons for you). Most windows also have a *title bar* above the content area, and a *border* surrounding both the content area and title bar.

The title bar is the center of control for the window. It holds the window's title, if it has one, and may contain buttons that can be used to dismiss it from the screen. Users move a window by dragging it by its title bar.

Windows with a title bar may also have a *resize bar* at the bottom, below the content area but within the border. By dragging any of the regions of the resize bar, the user can alter the size and shape of the window. Resizing is the only window control located outside the title bar.

The parts of a window are illustrated in Figure 2-8.

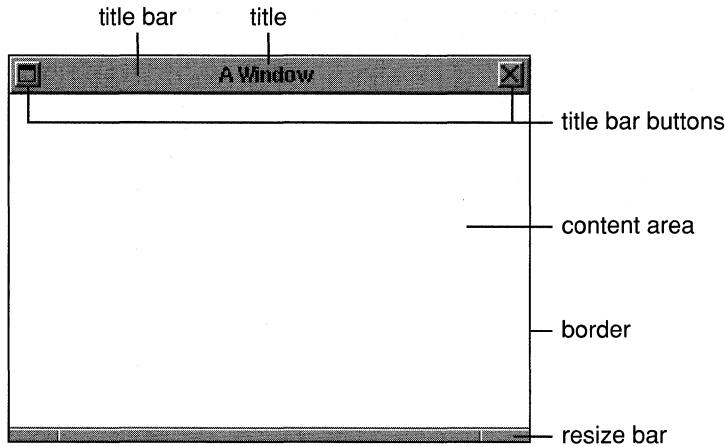


Figure 2-8. A Window

Visually, the various types of windows are distinguished by their size, by the way they're positioned relative to other windows, by the nature of the display within their content areas, and by their *style*—the appearance of their borders, title bars, and resize bars.

Window Style

Standard windows and panels come in three distinct styles:

Plain window	No border or title bar.
Titled window	A title bar and a black border. On the MegaPixel Display, the border is one pixel wide. The title bar is 21 pixels high and is separated from the content area by a one-pixel line that matches the window border. A window with a 100-by-100 pixel content area will be 102 pixels wide and 124 pixels high.
Resizable Window	A resize bar, title bar, and a black border surrounding both. The title bar and border are the same as for a titled window; the resize bar extends from the left to the right border at the bottom of the window. On the MegaPixel Display, the resize bar is eight pixels high. The top row of pixels is dark gray, the row beneath is white, and the remaining six rows are light gray.

Figure 2-9 illustrates resizable standard windows and Figure 2-10 illustrates titled panels. The different shades of gray shown in their title bars indicate their current status—whether they might be affected by user's next action. Window status is discussed under "The Key Window" and "The Main Window" later in this chapter.

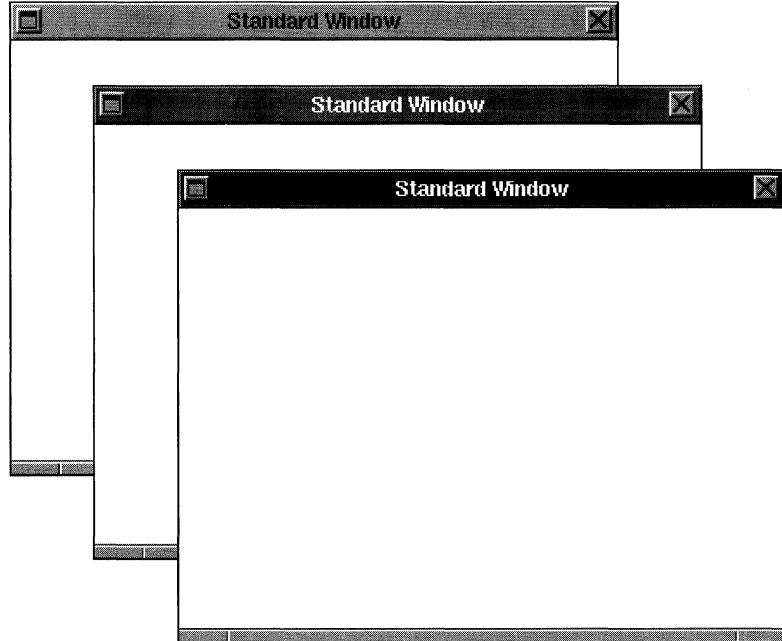


Figure 2-9. Standard Windows

Standard windows don't have to have a resize bar, though one is usually provided if the window displays scrollable contents.

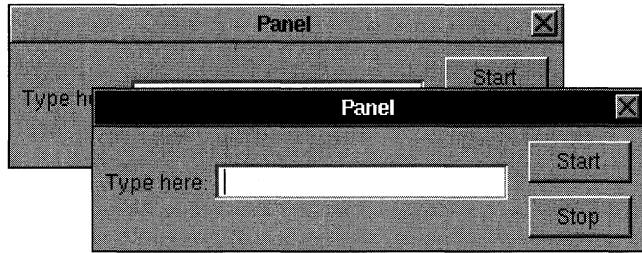


Figure 2-10. Panels

Panels can also be resizable, though they rarely need to be.

Like other panels, attention panels have a title bar and border, but the title bar remains empty, without a title or buttons. Because it doesn't permit the user to continue working in any other window of the application, it's important that an attention panel be immediately recognizable. The appearance of attention panels is discussed and illustrated under "Panels" later in this chapter.

Each of the other window types has its own style:

- | | |
|-------------|--|
| Menus | The title bar of a menu is solid black. The title itself isn't centered like the title of a panel or standard window; it begins at the left margin of the title bar. On the MegaPixel Display, menus have a dark gray border just on their top and left sides; the border on the bottom and right is drawn within the content area as part of the menu commands. Menus are illustrated in Figure 2-11. |
| Lists | Pop-up and pull-down lists have the same style as menus, except that they don't have title bars. |
| Miniwindows | Miniwindows lack borders and title bars, though miniature title bars and beveled borders are ordinarily drawn within their content areas. |
| Icons | Like miniwindows, freestanding and docked icons lack borders and title bars, but typically have beveled borders drawn within their content areas. |

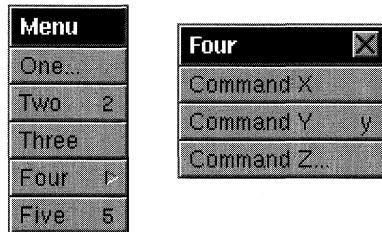


Figure 2-11. Menus

Conventions

Conventions govern the appearance of all windows. For standard windows and panels, the conventions are pretty much confined to the question of style. Title bars, resize bars, and borders have a fixed appearance, but you're free to draw within the window's content area as you see fit (though some restrictions apply to attention panels—see "Panels" later in this chapter).

However, the drawing done within the content areas of other window types is more constrained. Menus and lists, for example, enclose a group of commands, and each command must follow a specified format (see "Menus" later for details). Miniwindows and icons are even more tightly constrained.

Minindows

Minindows have a fixed size—64 pixels by 64 pixels on the MegaPixel Display. A small title bar is drawn across the top of a miniwindow, but it's inside the content area, not above it. The title bar shows the title of the window that was miniaturized, or at least a portion of it. Like icons, minindows have a beveled border, also drawn inside the content area, to make them look raised from the surface of the screen.

If a standard window holds an editable document, its minindow displays the icon for that document. In Figure 2-12, the minindow on the left shows the icon for a WriteNow document. This is the same icon that's displayed for the document in a directory window. The minindow to its right has the generic icon for ASCII documents.

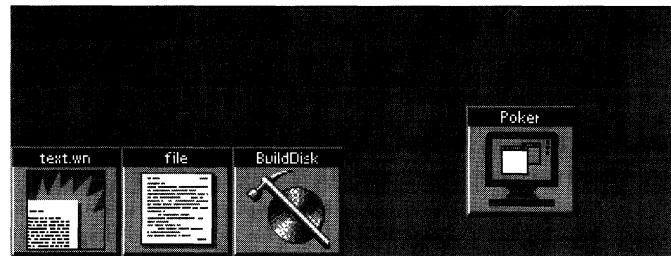


Figure 2-12. Miniwindows

If a standard window doesn't hold a document, its minindow displays the application icon. This also is illustrated in Figure 2-12. One minindow shows the icon for the BuildDisk application and another, labeled "Poker," shows the generic application icon.

Icons

Freestanding and docked icons are the same size as minindows and have the same beveled border to make them look raised above the screen. Each icon window displays the icon for an application—an image that on the MegaPixel Display can be no more than 48 pixels high by 48 pixels wide, so that it can fit comfortably in the 64 pixel by 64 pixel window.

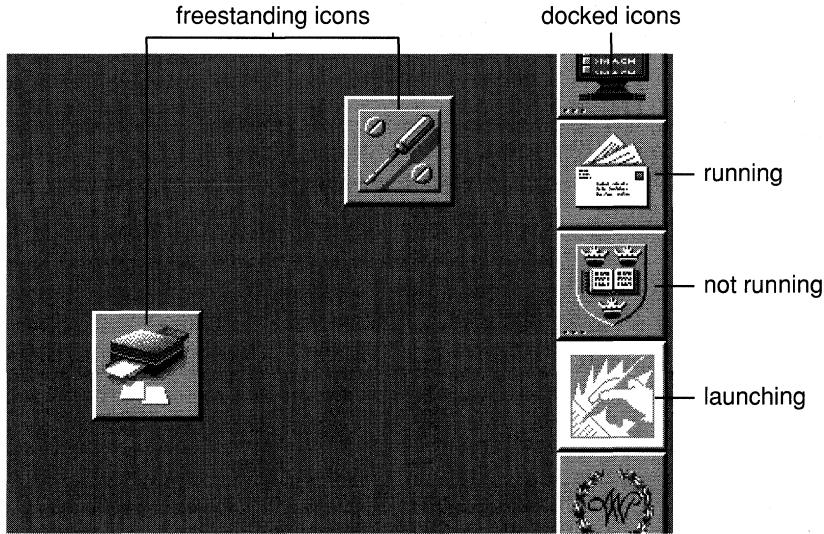


Figure 2-13. Freestanding and Docked Icons

Workspace Manager adds three small dots—similar to an ellipsis—in the lower left corner of a docked icon when the application the icon represents isn't running. The ellipsis disappears when the application is launched. While it's launching, the icon is highlighted in white, as shown in Figure 2-13. (Since freestanding icons appear on-screen only when the application is launched, they're never marked by an ellipsis.)

The docked icon for an application that isn't running displays the same image that represents the application in a directory window. However, you can provide an icon for the application when it's running that's different from, but related to, the icon found in a directory window.

Lists

Pop-up lists are used in lieu of a series of mutually-exclusive switches. They save screen space and prevent overcrowding in panels. Each list is controlled by a button that can be recognized by a special symbol,  , as illustrated in Figure 2-14. The label on the button that precedes the symbol indicates the current selection from the list. When the user makes a new selection, the button label changes.

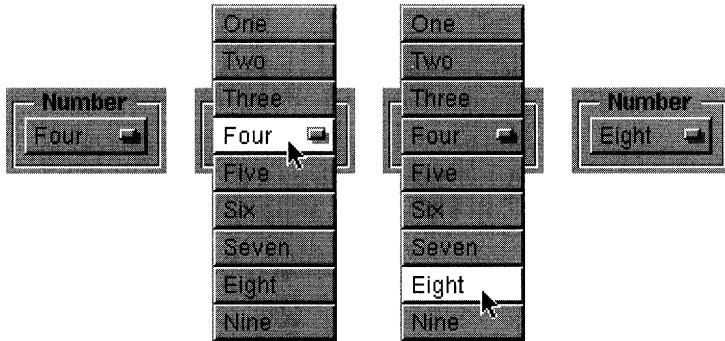


Figure 2-14. Pop-Up List

Pressing the button pops the list up so that the item matching the button label appears on top of the list. The list remains up only while the user holds the mouse button down. When the user releases the mouse button after dragging to a different item in the list, the label on the button changes to that item.

Since a pop-up list needs a more constant title than the changing button label, one is provided outside the button. In Figure 2-14, the list is entitled “Number.”

Pull-down lists are similar to pop-lists, except that they appear under the controlling button rather than on top of it. The button’s label doesn’t change and it’s marked by a special symbol, ▾, as illustrated in Figure 2-15.

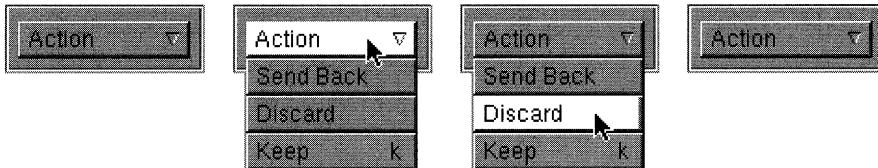


Figure 2-15. Pull-Down List

While the choices a user makes in a pop-up list set a state, the choices made in a pull-down list perform actions, much like menu commands. Before putting a pull-down list in your application, be sure that a menu wouldn’t do the job better.

Window Size

The only windows that have a fixed size are miniwindows and icons. The initial size of all other windows is determined by the application. Generally, standard windows are larger than panels and panels are larger than menus, but there are no fixed rules.

Window Ordering

Windows on-screen are ordered from front to back. Like sheets of paper loosely stacked together, windows in front can overlap, or even completely cover, those behind them. Each window has a unique position in the order. When two windows are placed side-by-side, one is still technically in front of the other.

Note: Even though a window is totally obscured by other windows, it's still considered to be on-screen; it retains its ranking in the order and can be exposed by moving the windows on top to the side.

To prevent menus and docked icons from being buried under larger windows, they're ordered above all others. Because attention panels demand, as their name implies, the user's full attention, they too are kept in front. On-screen windows are divided into these six *tiers*:

- Pop-up and pull-down lists are assigned the top tier. They remain on-screen only while the user holds a mouse button down so they only momentarily obscure other windows. Putting them in the top tier guarantees that they won't pop up in back of another window.
- Attention panels are assigned to the second tier. Like lists, they're only temporarily on-screen. But, unlike lists, the user must do something to dismiss them, rather than continue an action to keep them visible. Keeping an attention panel in front, where it can't be covered by other windows, confronts the user with it until it's dismissed and thus encourages prompt user action.
- The main menu is assigned the next tier back. In the absence of an attention panel or list, the usual case, it's the frontmost window on-screen.
- Other menus are assigned to a tier just below the main menu. They can cover each other, but not the main menu.
- Docked icons occupy the fifth tier. They can be covered by lists, attention panels, and menus, but not by the ordinary windows of your application.
- All other windows are grouped in the sixth—the last and largest—tier. Most of the windows seen on-screen are in this tier. They can cover each other, but can't come in front of the dock, menus, attention panels, or lists.

This six-tier system keeps attention panels, menus, and docked icons in view, and thus readily available to the user; it prevents them from being inadvertently lost in a large pile of windows. Although attention panels, menus, and docked icons may cover other windows, the user can get them out of the way when needed. Menus can be moved to the side or closed; the dock can be slid mostly off-screen. Attention should be attended to and dismissed.

When a window is first placed on-screen, it should come up at the front of its tier.

Window Placement

The user is free to rearrange windows on the screen, but the initial placement of windows is up to the application. To ensure a consistent user interface, all applications should follow these guidelines for locating windows:

- When an application is launched, its main menu should appear in the upper left corner of the screen. Users who prefer a different default location for the main menu can choose one with the Preferences application, and you may want to let users specify a preference just for your application.
- Standard windows should come up to the right of the main menu, allowing enough room for submenus that might later be attached to the main menu. Some applications also allow room for panels to come up to the left of the standard window and below the main menu.
- Attention panels should come up in the center of the screen, where they won't be overlooked.
- No part of any window (other than miniwindows and icons) should be placed off-screen.

You're encouraged to let users specify where the standard windows, main menu, and important panels of your application should appear the next time the application is launched. See the reference to a Preferences command under "The Main Menu" later in this chapter.

The Application Kit and Workspace Manager determine the initial placement of miniwindows and freestanding icons. They're lined up along the lower edge of the screen to keep them out of the way of other windows.

Window Behavior

Windows respond to user actions in the following ways:

- Any window can be brought to the front of the screen, relative to other windows in its tier.
- Any window with a title bar can be moved to a new location on the screen, as can any icon or miniwindow.
- Any window with a resize bar can be resized.
- A window with the appropriate buttons in its title bar can be closed or miniaturized.

A standard window can display either of two buttons in its title bar:

Miniatrize button Clicking the miniaturize button replaces the window with its miniwindow counterpart. The miniwindow represents the window on-screen and gives the user access to it; double-clicking the miniwindow causes it to disappear and the miniaturized window to reappear.

Close button Clicking the close button removes the window from the screen.

When the user clicks in a title bar button, the action of the button is performed. The click doesn't count as "clicking in a window" for the purpose of bringing the window to the front, making it the key window, or activating an application (the key window and active application are discussed in the "Application and Window Status" section of this chapter).

Title bar buttons are illustrated in Figure 2-16. The window on top has both buttons as they normally appear. The miniaturize button is on the left and the close button is on the right. The window in back shows what the close button looks like when the window displays a document that the user has edited but hasn't saved.

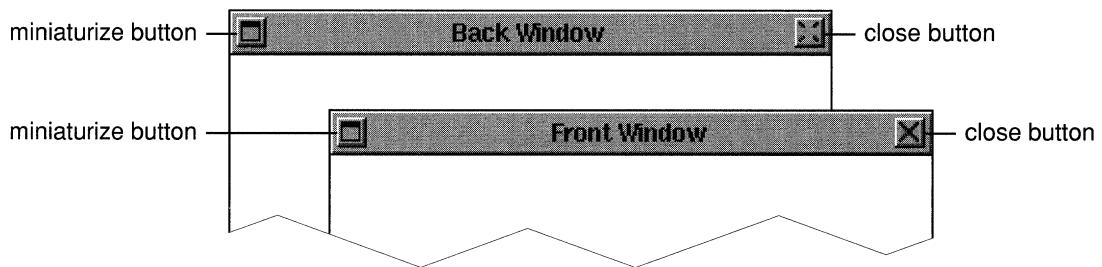


Figure 2-16. Title Bar Buttons

Reordering Windows

Clicking in a window brings it to the front of its tier, provided that the click isn't in a title bar button. The window is reordered immediately as the mouse button is pressed. Unlike clicking a button or menu command, the action doesn't wait until the mouse button is released. If the user is dragging the window to a new location, this lets the window assume its reordered position before being moved.

Because the main menu is the only window in its tier, its order relative to other windows can't be changed. Docked icons similarly share a unique tier; since they never overlap, reordering them is pointless.

Moving Windows

The user can drag any window by its title bar. The action of pressing and releasing the mouse button to drag the window also counts as a click and brings the window to the front of its tier.

Icons and miniwindows don't have title bars but can be dragged from any point within their interiors. A docked icon can be dragged to a new position in the dock, but it disappears if left anywhere else, unless the application it represents is running. An icon not in the dock can be dragged anywhere on the screen just like other windows. But if it's left near an empty slot in the dock, it may snap into the slot.

The dock itself can be dragged by the NeXT logo. It moves up and down in its track at the right of the screen, but can't move to the left or right. No part of the NeXT logo can be dragged off-screen.

When a window is dragged, one point within it remains aligned with the mouse cursor. Since the cursor always stays on-screen, the point aligned with the cursor also remains on-screen. This prevents the window from being dragged off-screen to a location where the user can't retrieve it.

Resizing Windows

A window with a resize bar can be resized by the user. As the user begins dragging in the resize bar, an outline of the window edge snaps to the cursor. The outline follows the cursor until the user releases the mouse button to end dragging. The window then resizes to the outline.

When the user begins dragging in the central region of the resize bar, the window becomes shorter or taller depending on whether the user drags up or down. Beginning at either end of the resize bar and dragging to the left or right causes the window to become wider or narrower. Beginning at either end and dragging diagonally moves a corner (and its two adjacent sides) in or out.

An application can constrain the shape of a resizable window so that it doesn't become too big or too small, or so that it grows and shrinks in unit amounts.

Note: The window outline snaps to the cursor only after the user starts dragging. This prevents a click in the resize bar—which might be meant only to bring the window to the front—from inadvertently resizing it.

Closing Windows

The close button removes a window from the screen. What this means depends on the type of window:

Menus and panels A menu that's closed is removed from the screen, but the user retains a way to quickly retrieve it through a command in another menu. Panels that are closed are retrievable in the same way. (See "Menus," later in this chapter, for more information.)

When a panel that was closed is returned to the screen, it assumes its former size and location, and it retains its former state. From the user's point of view, and programmatically, it's the same panel that was closed.

Standard windows Closing a standard window usually removes it from the application as well as from the screen. From the user's point of view, the same window can't again be made visible. The application might create a new window with the same title and a similar display, but there would be differences. The selection might not be preserved, and the new window won't necessarily be located in the same place or have the same shape as the old one, especially if the user had moved or resized the window that was closed.

Closing a window that displays the contents of a file normally also closes the file. Users should be given a chance to save any changes to the file before it's closed. This is typically done through an attention panel such as the one illustrated in the "Panels" section later in this chapter.

The close button has a matching command in the Window menu, and the command has a keyboard alternative, Command-w (for "window"). Command-w closes a window selected by the user, if the window has a close button. (See "The Window Menu" below for details.)

Miniaturing Windows

Miniaturing a window removes it from the screen without destroying it or its contents. The window disappears from view, and a miniwindow appears to represent it on-screen. Double-clicking the miniwindow reverses the miniaturization.

Users can't work in a miniaturized window, but programs can continue to alter its display. For example, if you begin compiling a program in a Terminal window, and then miniaturize the window, you'll see any error messages written by the compiler when you return the window to the screen.

Miniaturizing a window differs from closing one in a number of ways:

- Miniaturizing preserves the window as it was last seen on-screen; a window that's closed can't be retrieved in the same state.
- Miniaturizing a window leaves behind a miniwindow so that it can be brought back to the screen; closing a window doesn't provide the user with a way of getting it back.
- Miniaturizing a window that displays a file won't close the file or change the way it's displayed; closing a window closes the file it displays.

Menus can't be miniaturized and don't need to be, since closing serves roughly the same function. When a menu is closed, it's preserved and can be returned to the screen through a menu command; a miniwindow isn't needed to get it back. Menus are generally small to begin with; there wouldn't be much advantage to miniaturizing them further.

A panel can have a miniaturize button, but it's rare that one would be needed. Like menus, panels can be closed and returned to the screen through a menu command. A miniaturize button is redundant, unless a miniwindow would, for some reason, be more convenient for the user than the command, or unless the panel persists on-screen when the application isn't active (see "Persisting Panels" later in this chapter).

Like the close button, the miniaturize button has a counterpart command in the Window menu that can miniaturize a target window. But, unlike the close button, its command doesn't have a standard keyboard alternative. See "The Window Menu" later in this chapter for information on how the command works.

Hiding and Retrieving Windows

The Hide menu command and its keyboard alternative, Command-h, let the user clear the screen of all the windows belonging to an application. This opens up the workspace so that it's easier to work in another application.

When an application is hidden, only its freestanding or docked icon remains on-screen. When the user double-clicks the icon, the hidden windows reappear at the front of the screen. Users can resume working in the application, picking up again at exactly the same point where they left off.

Double-clicking an application icon has one other effect: It activates the application (as discussed in the next section), and so may cause the menus and panels of another application to disappear while those of the newly activated application reappear.

Double-clicking the icon for a running application always activates it and brings its windows to the front, even if the application wasn't hidden. A hidden application has all its windows return at the front of the screen. However, if the application wasn't hidden, but its windows are simply covered by the windows of other applications, double-clicking its

icon brings only its panels and frontmost standard window forward. Other standard windows may remain covered. Since the double-click activates the application, its menus also return to the screen.

Note: A window that's completely obscured by other windows is "covered," but not "hidden" in the sense used here. A covered window can be made visible by moving the windows on top of it to the side. A hidden window can't be; it's completely removed from the workspace.

Application and Window Status

Since UNIX can run more than one application process at a time, the screen is likely to display windows for a variety of different applications. Workspace Manager is one application that will almost always have a window on-screen. Some users will also run Mail and a spreadsheet, or perhaps a word processor and Digital Webster, at the same time as other applications.

The user must be able to pick a particular application, and a particular window in that application, to work in. The application that the user is currently working in is known as the *active application*; the windows that are the current focus of user attention in the active application are the *key window* and the *main window*. The key window and main window are usually one and the same; the two terms identify different functional roles that can be assumed by the same window:

- The key window is the window that receives characters from the keyboard.
- The main window is the window containing the selected target for control actions.

These three concepts—the active application, key window, and main window—refer not to inherent properties of applications and windows, but to their status at a particular point in time. They're discussed more fully in the three sections that follow.

The Active Application

Out of all running applications, at most one is selected to be the *active application*, the principal application the user is working in. An application must be activated—made to be the active application—before the user can type in its windows or use its menus.

The active application differs from other running applications in four ways:

- It's the only application with visible menus. When an application is deactivated, its menus are hidden from view; when it's reactivated, they're restored to the screen.

- It's the application that owns most, if not all, of the panels that are visible on-screen. In general, panels behave like menus; they hide when the application isn't active and return to the screen when the application is reactivated. In exceptional cases, however, the application may choose to leave a panel on-screen even when the application isn't active. (See "Panels" below for guidelines on when it's appropriate to allow a panel to persist.)
- It's the application that receives the user's keyboard actions. Typing and keyboard alternatives can affect only the active application. When there's no active application, the user's keystrokes have no effect.
- It's the application that contains the key window and main window (if there is a current key window or main window), and its windows are likely to be in front of the windows of other applications.

Activating an Application

The task of selecting the active application is left to the user. An application never becomes active unless the user does something to activate it. The user's action can be direct, such as launching the application or clicking in one of its windows, or indirect, such as having one application send a message to another application.

An application is activated when:

- The user launches it, provided no other application is currently active. After launching an application, Workspace Manager deactivates itself so the newly launched application is free to become active. Unless the user reactivates Workspace Manager or activates another application while the newly launched application is being read into memory, the new application will become the active application.
- The user double-clicks a miniwindow belonging to the application, or double-clicks the application's freestanding or docked icon. Double-clicking a docked icon will launch the application if it's not already running.
- The user clicks within one of the windows belonging to the application, provided the window isn't a miniwindow or icon. (Clicking a miniwindow or icon just once may bring it to the front, but won't activate the application.)
- It receives a message from another application, if the message asks it to do something that may require interaction with the user. A message to open another file received from Workspace Manager is one such message. A message sent to Webster asking it to define a word is another. (See "Conditional Activation" below for details.)

A docked application that's launched automatically when Workspace Manager starts up is not activated; Workspace Manager remains the active application. Only applications that are launched due to direct user action are given the opportunity to become active.

Deactivating an Application

There can be only one active application per workspace (that is, one per Window Server) at a time. Whenever the user chooses a new active application, the previous one is automatically deactivated. The Application Kit and Workspace Manager take care of this task.

However, some user actions can deactivate an application without activating a successor. The active application is deactivated when:

- The user hides its windows (by using the Hide command).
- The user terminates it (by choosing the Quit command).

In neither case is another application activated. It takes a positive act on the user's part to make an application active.

In addition, an application should deactivate itself just before sending a message to another application, if the intent of the message is to have the other application become active. (See "Conditional Activation" below for details.)

Note: A deactivated application is still an active process running on the computer. It's "deactivated" only in the sense that it no longer is the active application.

Conditional Activation

Applications communicate with each other through messages. When an application receives a message asking it to do something that might require user participation—even to the extent of operating a scroller—it needs to become active. However, it should activate itself only if the user hasn't turned to another application and made it the active application.

For example, one application could send a message requesting the services of another application (as a word processor might call upon a spelling checker). The first application (the word processor) deactivates itself immediately before sending the message to the second application (the spelling checker). The second application then activates itself on condition that no other application is currently active. Since the first application had deactivated itself, this condition will be met, unless the user has activated another application in the meantime.

Conditional activation follows from the principle of user control. This principle is violated if an application forces activation when the user wants to turn to something else.

The Key Window

Users expect to see their actions on the keyboard and mouse take effect not only in a particular application, but also in a particular window of that application. Each user action is associated with a window by the Window Server and Application Kit. Before acting, the user needs to know which window will be affected; there should be no surprises.

Since the mouse controls a cursor, it's quite easy for the user to determine which window a mouse action is associated with; it's whatever window the cursor is pointing to. But the keyboard doesn't have a cursor, so there's no natural way to determine where typing will appear.

The window associated with keyboard actions, the one where typing will appear, is known as the *key window*. To mark the key window for users, the Application Kit highlights its title bar (by turning it black). A window without a title bar can also be the key window, but only windows with title bars are marked. A potential key window should always have a title bar.

Key window highlighting is illustrated in Figure 2-17.

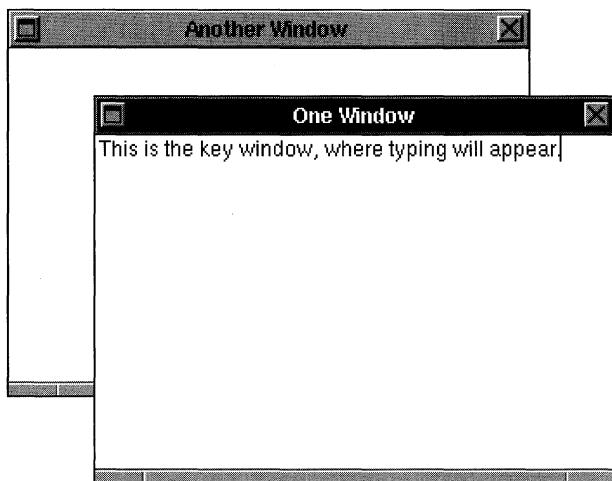


Figure 2-17. The Key Window

You can think of the highlighting as a kind of cursor for the keyboard. It shifts from window to window as the key window changes. Key-window status also moves from application to application as the active application changes. Only one window on the screen is marked at a time, and it must be in the active application. There's just one key window per Window Server—that is, only one per machine and keyboard.

For a window to be designated as the key window, it must be able to accept characters from the keyboard. Standard windows should always have this ability, but menus lack it, as do lists, miniwindows, and icons. Each application can determine which of its panels can receive keyboard actions. (See Chapter 5 and Chapter 7, "Program Dynamics," for information on how to do this.)

Note: A window doesn't have to become the key window to receive, and act on, keyboard alternatives. It does, however, have to be in the active application.

Since the key window belongs to the active application, its black title bar has the secondary effect of helping to show which application is currently active. The key window is the most prominently marked window in the active application, making it "key" in a second sense: It's the main focus of the user's attention on the screen.

You probably want the principal windows of your application to receive this attention; any that contain documents or hold the application's principal display should certainly receive it. In general, all the standard windows in your application should be permitted to become the key window, even if they don't respond to keyboard actions. Giving key window status to a window focuses attention on it and prevents the user from typing in any other window.

If the key window doesn't do anything with the user's typing, it should beep as it receives the keystrokes to indicate to the user that typing isn't appropriate.

The Main Window

The standard window where the user is currently working is known as the *main window*. Usually, the main window is also the key window, and therefore the main focus of attention on the screen. Whenever a standard window becomes the key window, it also becomes the main window.

Panels and menus can never be the main window; only standard windows can. But many panels can be made the key window. When key window status shifts from a standard window to a panel, main window status remains with the standard window.

The main window is the focus of user actions in panels and menus. The Find panel in WriteNow, for example, requires the user to supply information by typing it. Since the panel is the destination of the user's keystrokes, it's marked as the key window. But the panel is just an instrument through which users can do work in another window—the main window.

So that users can pick out the main window when it's not the key window, the Application Kit highlights its title bar in dark gray. If the main window is also the key window, it has only the black highlighting of the key window.

Figure 2-18 illustrates the main window when it's marked as the key window and when it's not.

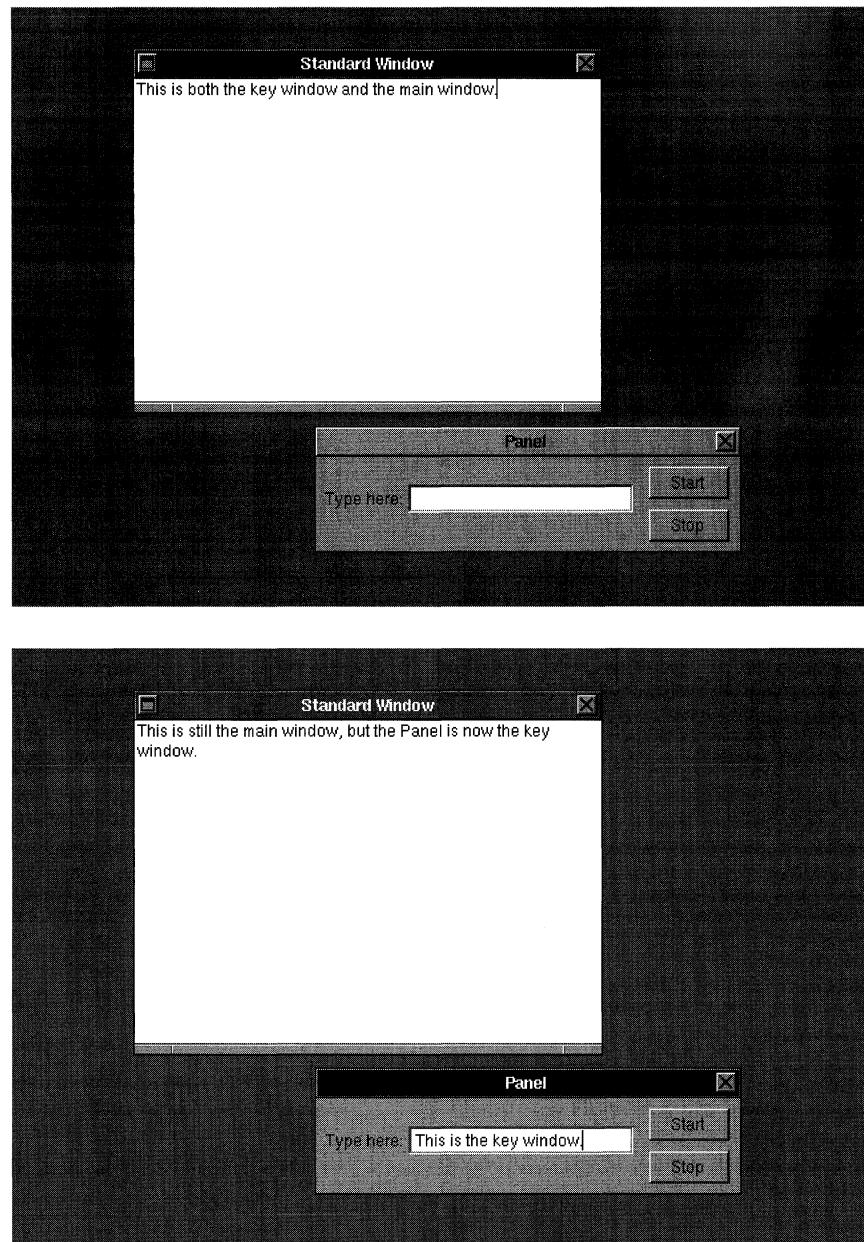


Figure 2-18. Marking the Main Window

Like user actions in a panel, menu commands may affect the main window. For example, the Save command saves the document displayed in the main window and the Bold command turns the current selection in the main window bold.

A menu command can also affect a panel. A Paste command could enter text in a Find panel, for example. For this reason, user actions in a panel or menu need to be associated with both the key window and the main window:

- An action is first associated with the key window.
- If the key window is a panel and it can't handle the action, the action is next associated with the main window.

Note that this order of precedence is reflected in the way windows are highlighted: The key window is always marked; the main window is marked only when it's not the key window.

The main window is always in the same application as the key window, the active application. It follows the key window as the user's actions shift the focus from window to window and from application to application.

Choosing the Key Window and Main Window

Whenever possible, the user, rather than the application, selects the key window and main window.

In the Active Application

In the active application, the user can select a new key window by clicking in it. If the window is a standard window, it's also made the main window. If it's a panel, it's highlighted as the new key window, but the former main window retains its status and is highlighted in dark gray. The user can't make a window the main window without also making it the key window.

The Application Kit chooses a new key window (or main window) for the active application whenever the user closes or miniaturizes the window currently having that status:

- If the closed or miniaturized window was the key window but not the main window, the Application Kit makes the main window the new key window.
- Otherwise, the Application Kit chooses the frontmost window that accepts typing to be the new key window, and the frontmost standard window to be the new main window. The new key window will generally also be the new main window.

When an Application Is Activated

When an application is activated, one of its windows is made the key window and one (perhaps the same one) is made the main window. Again, whenever possible, the user makes the selection:

- If the user activates the application by clicking in a window that accepts keystrokes, it becomes the key window. If the window is a standard window, it's also made the main window.
- If the user activates an application by double-clicking a miniwindow, the window it represents again appears on-screen and becomes the key window and main window.

If an application is activated without the user directly selecting a new key window, the user's previous selections are honored. For example, if the user reactivates an application by double-clicking its icon, the previous key window and main window are restored.

When an application is activated on launch, there are no previous choices to honor. The application should designate one of its windows to be the initial key (and main) window. If the application opens a document file for the user, the window that displays the document should be the key window.

Note: When a new application is activated, its key window may be highlighted before the former key window (in the deactivated application) loses its highlighting. This is an unavoidable consequence of a multitasking environment. Users can begin working in one process (the new active application) before their instructions to another process (the previous active application) have been completed. Although the former key window may retain its highlighting for a short time, it's no longer the key window; all keyboard actions are associated with the new active application.

Clicking in a Window

Clicking in a window has two separate, but related, results:

- The window becomes the key window (and usually also the main window), and its application is activated.
- The window comes to the front of its tier.

The first is a change in the window's status, the second in its position on-screen.

It's clear that both results are required to make the window available to the user to work in. The window needs to be reordered in front of other windows so that its contents aren't covered. It also must become the key window for the user to be able to type in it and for it to receive menu commands. For a window to become the key window, its application must be activated.

On the NeXT computer, however, these two results of a mouse click, while logically related, are not inseparable. If the click is in the window's title bar and is modified by the Alternate key, it brings the window to the front, but doesn't make it the key window or activate its application. Alternate-clicking in the title bar thus lets users rearrange and reorder windows on the screen without changing the current key window, main window, or active application.

Working in a Window

Users also click to work within windows—to operate buttons and sliders, to scroll, and to make selections. If the user clicks in a window that isn't already the key window, a question arises concerning intent: Did the user intend the click just to bring the window forward and make it the key window, or was it also intended to do some work within the window? This question is addressed by the following guideline:

- If the user chooses a particular control—such as a button or scroller—to click in, the click will not only bring the window forward, make it the key window, and activate its application, the click will also operate the control. Since controls are small, it's reasonable to assume that the user chose to click the control, not just the window.
- If the click is just generally within the content area of the window, the click will bring the window forward, make it the key window, and activate its application, but won't do anything else. Specifically, the click won't alter the current selection.

However, if the user chooses to double-click within the content area of the window, the double-click will act just like a double-click in the key window. Double-clicking on a word should select the word whether the window is the key window or not.

Making a Click Unnecessary

When an application designates a new key window for the user to work in, as it does when the current key window is closed or miniaturized, it simulates the action of a mouse click by bringing the window forward. Whenever an application acts on the user's behalf, its action should do exactly what the user's action would have done.

Menus

A menu is a titled window that displays a vertical list of commands; each command controls a particular action that's carried out when the command is chosen.

To choose a command, the user presses the left mouse button as the cursor points anywhere within the content area of the menu and releases it as the cursor points to the desired command. This can be as simple as clicking the command, or the user can drag through the menu, from command to command. Each command that comes under the cursor while the mouse button is down is highlighted.

Menus provide users a point of entry for all the functionality of an application, its obscure and common features alike. Because of this special role, they behave in a special way:

- All the visible menus for an application disappear when the user starts working in another application. They reappear when the user returns to the application. (Menus that weren't previously on-screen don't reappear.)
- Menus are segregated into two of the frontmost tiers of on-screen windows. They appear to float above all other windows (except attention panels and lists).
- Menus can't be miniaturized. They don't need to be, since closing a menu serves only to hide it temporarily.
- Menus are hierarchically arranged. Choosing a command in one menu can produce another menu with its own list of commands.

The first three of these points were discussed earlier in this chapter. (See “The Active Application,” “Window Ordering,” and “Miniaturizing Windows” above.) The menu hierarchy is discussed below.

Submenus

Every menu, except the main menu, should be a *submenu* of another menu, its *supermenu* in the application's hierarchy of menus. The main menu is at the top of the hierarchy; it's the only menu that's not a submenu.

Each submenu is associated with a particular command in its supermenu. The submenu typically remains off-screen until the user chooses the command that it's associated with. This makes the submenu visible and *attaches* it to its supermenu.

An attached submenu is displayed alongside and to the right of its supermenu. The title bars of the two menus are aligned at the same height. Figure 2-19 shows a main menu with its Edit submenu attached.

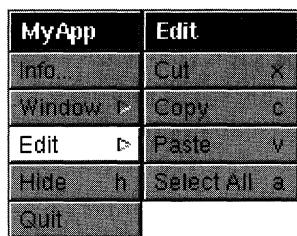


Figure 2-19. Attached Submenu

The submenu comes to the screen as soon as the user presses the mouse button or drags into the controlling command. The attachment is usually temporary, however; the submenu disappears when the user drags into another command. The user can drag from a controlling command into a submenu to choose one of the submenu's commands. As long as the mouse button is held down, the submenu remains visible and the controlling command stays highlighted.

Keeping a Submenu Attached

If the user brings a submenu to the screen and releases the mouse button while the submenu is still visible, the submenu remains on-screen and attached to its supermenu. The easiest way to attach a submenu is simply to click its controlling command, but the user can also drag to the command and release the mouse button while the submenu is visible.

There's just one exception to this rule: If the user drags into the submenu to choose one of its commands, the submenu disappears once the choice has been made. The intent of the user's action is to choose a submenu command, not to attach the submenu and keep it on-screen.

The controlling command for an attached submenu stays highlighted to indicate that the submenu is attached. In Figure 2-19 above, the Edit command is highlighted while the Edit menu is attached.

A supermenu and its attached submenu act like a single window. User actions that move or close the supermenu also move and close the submenu; an attached submenu has no close button of its own. A submenu attached to the main menu is assigned to the same window tier as the main menu.

An attached submenu can also have its own submenu attached. This is illustrated in Figure 2-20. The Sink menu is attached to Kitchen, and Kitchen is attached to House. Moving or closing the House menu serves to move or close all three.

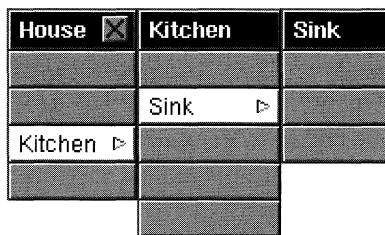


Figure 2-20. Three Attached Menus

Tearing off an Attached Submenu

The user can *tear off* an attached submenu by dragging it away from its supermenu. Moving it free of its supermenu gives it an independent life on-screen. As a sign of its independence, it gets, for the first time, its own close button. The close button identifies the menu as a torn-off submenu. (Any submenus that were, directly or indirectly, attached to the torn-off submenu move with it and remain attached.)

The idea is for users to attach a submenu, then tear it off and move it to a desired location if they want it to stay on-screen. Once a submenu has been torn away from its supermenu, it stays where the user puts it.

When the user drags through a menu, copies of torn-off submenus are temporarily attached while the cursor is over their controlling commands. It's possible to drag into the copy to choose a command, just as for any other temporarily attached submenu. Figure 2-21 shows a transient copy of a torn-off Edit menu.

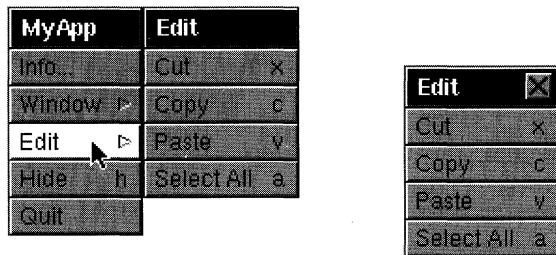


Figure 2-21. Attached Copy

The torn-off submenu stays in place while the copy is on-screen. The copy is temporary; it stays on-screen only while the mouse button is held down. When the mouse button is released, the copy disappears, even if the cursor is over the controlling command. To reattach a torn-off submenu and keep it attached, the user must first close it.

Detaching a Submenu

Besides tearing it away from its supermenu, users can detach a submenu in any of three other ways:

- By again choosing its controlling command. This removes the submenu (and any submenus attached to it) from the screen. Choosing the Kitchen command in the House menu in Figure 2-20 above detaches both the Kitchen and Sink submenus.
- By choosing any other command in the supermenu.
- By closing the supermenu. When a supermenu closes, its attached submenu is detached and closed.

All three ways of detaching a submenu remove it from the screen; it disappears from view. If the submenu has its own attached submenu (as Kitchen does in Figure 2-20), they both disappear (and both detach).

Submenu Hierarchy

Since menus should be easily accessible to the user, you should try to keep your application's menu hierarchy as shallow as possible. In general, a menu should be located no more than two steps away from the main menu. (Note that, for purposes of illustration, Figure 2-20 violated this guideline. Since the House menu was torn off from another menu, the Sink menu is at least four steps away from the main menu.)

A menu can have as many submenus as it has commands (but only one can be attached at a time). A menu can also be made the submenu of more than one menu, but there would be little reason to establish this relationship and it's not recommended.

Commands

A menu can display several different kinds of commands. Some act like controls in the control-action paradigm—Hide, Quit, and Info, for example. Others—such as, Copy, Paste, and Miniaturize—participate in the target-selection paradigm.

Many commands cause other windows to appear on-screen:

- Some bring up a standard window—the New command in the Window menu, for example.
- Some control submenus. The action of the command is simply to attach the submenu to the menu. The command stays highlighted while the submenu is attached.
- Some put an attention panel on-screen to help clarify or complete the command. For example, the Save As command produces a panel that asks the user to type in the name of the file where the document should be saved. The controlling command (Save As) remains highlighted until the attention panel is dismissed.
- Others bring up a panel that can stand on its own, independent of the command that produced it. Sometimes the panel simply imparts information to the user—a Help panel, for example. But usually it acts as a control panel where the user can give instructions to the application—the Font and Find panels, for example. Such panels are very similar to submenus in that they open a range of options to the user. However, unlike submenus, they don't attach to a supermenu and can't have submenus (or subpanels) of their own.

Menu commands should be short, consisting of a single word if possible, a short phrase if not. If a command controls a submenu or brings up a submenu-like panel, it should be identical to the title of the submenu or panel. If it brings up a standard window, it should match the title of the window if at all possible. Otherwise, wherever possible, the first word of the command should be a verb, so the command reads like a short imperative sentence for the action it performs.

The first and last words of a command begin with uppercase letters; all other words are capitalized as they would be in a title. It's preferable to avoid abbreviations in commands, especially those that aren't standardized or widely used.

When a command can't be carried out, it should be dimmed (displayed in gray rather than black). For example, the Save command is dimmed when there's no open file to save. Dimmed commands shouldn't be highlighted in response to user actions.

Commands that control submenus are marked by the submenu symbol, , as illustrated in Figure 2-22. Those that control a panel are marked by an ellipsis (...). Commands that bring up a standard window (like the New command in the Window menu) are unmarked; they aren't followed by an ellipsis.

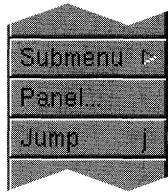


Figure 2-22. Sample Commands

The character used as the keyboard alternative for a command is displayed to the right of the command, aligned with the submenu symbol. Commands that control submenus can't have keyboard alternatives, although those that bring up a panel can.

When a command is activated by a keyboard alternative, the command is highlighted just as if it had been clicked. Keyboard alternatives can also operate commands in off-screen menus. When they do, the closest command in a menu on-screen is highlighted. For example, if the Kitchen and Sink menus in Figure 2-20 above were off-screen and the user chose one of the commands in the Sink menu, the Kitchen command in the House menu would be highlighted. This ensures users of immediate, visual feedback that the keyboard alternative has in fact invoked the command.

If a menu command controls a submenu, it remains highlighted as long as the submenu is attached. If it controls an attention panel, it remains highlighted until the panel is dismissed from the screen. Commands don't stay highlighted if they bring up a panel that's not an attention panel.

The Main Menu

Every application should have at least one menu, a *main menu* that bears the name of the application, or a suitable abbreviation, in its title bar. If an application has just a main menu, it holds all the commands for the application. If it has more than one menu, all but the main menu should be made submenus of another menu. Through the hierarchical arrangement of submenus, the main menu gives the user access to all the menus of the application.

Because the main menu is at the top of the menu hierarchy, it lacks a close button and always remains on-screen when its application is active.

Placement

When an application is first launched, its main menu appears in the upper left corner of the screen. This has several advantages to the user:

- Because every main menu comes up in the same place, users always know where to find it.
- The name of the current active application, in the main menu's title bar, is always in a common and logical location on-screen.
- In the upper left corner, the main menu is less likely to block the view of other windows. Standard windows are generally more centrally located on the screen, the dock adheres to the right edge of the workspace, and miniwindows and icons line up along the bottom edge.
- A rapid movement of the mouse easily brings the cursor to the main menu. Because the cursor can't leave the workspace, it stops within the menu; the user doesn't have to be concerned with moving the cursor too quickly or too far.

However, users can change the default location of the main menu with the Preferences application, and you're encouraged to permit users to state a specific preference for where the main menu of your application should come up the next time it's launched.

Bringing the Main Menu to the Cursor

If the right mouse button is enabled (with the Preferences application), it can be used to gain quick access to the main menu. When the user presses the right mouse button anywhere in the workspace (except over an icon), a copy of the main menu for the active application appears under the cursor. The copy stays on-screen until the right mouse button is released.

To begin, the cursor lies directly over the main menu's title bar. The user can drag down into the menu (and into a temporarily attached submenu) to choose a command. When the right mouse button is released, the copy and any temporarily attached submenus disappear.

Note: Users can also enable the left mouse button for this special function (see “Left and Right Orientation” earlier in this chapter). By convention, documentation takes the point of view of right-handed users, who generally prefer to enable the right mouse button.

Standard Commands

There's a great deal to be gained if commands shared by most applications are arranged similarly in similar menus. Every application should lay out its main menu like the model illustrated in Figure 2-23.

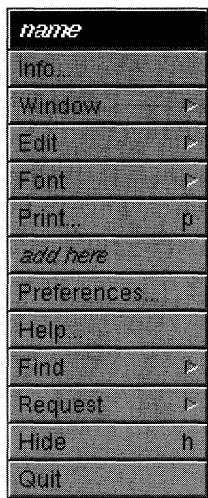


Figure 2-23. The Main Menu

The commands shown in this figure are described below:

Command	Action
Info...	Brings up a panel containing information about the application. It might display the name of its author, a copyright notice, a version number, or a few lines of help. You can determine the type of information most appropriate for your application. (See "The Information Panel," under "Panels" below.)
Window	Attaches the Window menu, which contains commands affecting windows and the files displayed in windows. See "The Window Menu," below.
Edit	Attaches the Edit menu, which contains commands affecting the current selection in editable documents. See "The Edit Menu," below.
Font	Brings up the Font menu, which has commands to alter the font of the current selection. Each command alters one aspect of the font, such as its size or style, while leaving other aspects intact. There's also a command to bring up the Font panel, one of the specific panels defined in the Application Kit. The Font menu and Font panel target currently selected text; the Preferences panel should be used to alter the default font for static displays. (See "The Font Menu" below.)
Print...	Brings up a panel that permits the user to print a document. This panel is also defined in the Application Kit.
Preferences...	Brings up a panel that permits the user to customize the application. All preferences should carry over to the next time the user launches the application; most will also affect the way the application works during the current session. Preferences can include such things as the location and size of windows, the default font size, the format for displaying data, and where to store document files.
Help...	Brings up a panel with helpful information on how to use the application, or with instructions on how to get that help.
Find	Attaches the Find menu, which contains commands related to the Find panel. One of its commands brings up the panel. See "The Find Menu" below.
Request	Brings up a menu with commands that request action from other applications. For example, users can request the dictionary application (Digital Webster) to look up the current selection, or Workspace Manager to open a file. See "The Request Menu" below.

Hide	Hides all the windows of the application. See “Hiding and Retrieving Windows” above.
Quit	Terminates the application. If the application has any open files that the user has altered but not saved, this command should bring up an attention panel giving the user the option of canceling the command and perhaps also of saving the altered files.

The Info, Hide, and Quit commands should be in the main menu of every application. The other commands shown in Figure 2-23 should be included where appropriate. Since almost all applications require a Window menu to hold commands corresponding to title bar buttons, the main menu almost always has a Window command too.

Other frequently used commands can be added to the main menu after the Print command and before Preferences. Added commands should pertain to the entire application. Less common commands should be placed in submenus.

The Window Menu

The Window menu contains commands affecting windows and the files they display as a whole. Commands affecting selected contents of the window are mainly in the Edit menu.

Figure 2-24 illustrates the standard format of the Window menu.

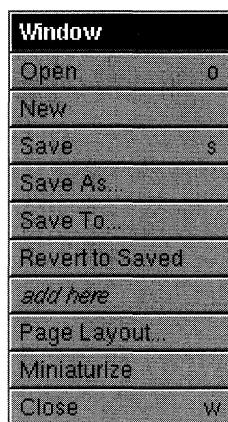


Figure 2-24. The Window Menu

The commands illustrated in Figure 2-24 are summarized below:

Command	Action
Open...	Brings up the Open panel so the user can open a file. Opening a file also opens a window to display it in. The Open panel is provided by the Application Kit.
New	Opens a new, unnamed file and a window to display it in.
Save	Saves changes to a file (writes them to the disk). If the file is unnamed, this command should have the same effect as the Save As command.
Save As...	Saves the file displayed in the window, as changed, by writing it to a new file with a name supplied by the user. The new file becomes the file displayed in the window, and the window's title is changed accordingly. This command places an attention panel on-screen which asks the user to type in a file name, or cancel the command. This panel is defined in the Application Kit.
Save To...	Saves the file displayed in the window, as changed, by writing it to a new file with a name supplied by the user. In this respect, Save To is identical to the Save As command. However, Save To doesn't replace the window's current file with the new one. You can choose whether to implement Save As or Save To or both in your application.
Revert to Saved	Replaces the current version of the file displayed in the window with the version saved on disk. This undoes any changes made to the file since it was last saved.
Page Layout...	Brings up the Page Layout panel, which lets users determine how documents are to be printed and displayed on the screen.
Miniaturize	Miniaturizes the current key window, if it has a miniaturize button. If the key window doesn't have a miniaturize button but the main window does, the command miniaturizes the main window. If neither window has a miniaturize button, the command has no effect. (See "Miniaturizing Windows" above.)
Close	Closes the current key window, if it has a close button. If the key window lacks a close button but the main window has one, the command closes the main window instead. If neither has a close button, the Close command is inoperative. (See "Closing Windows" above.)

In some applications, the user can open documents that are displayed in more than one window. For these applications, the Close command should be a counterpart to the Open command: It should close the document file and all the windows that display it. The Window menu should then include another command, Close Window, that would close a single window (if it has a close button). Close Window, rather than Close, has the Command-w keyboard alternative. The arrangement of these commands is illustrated in Figure 2-25.

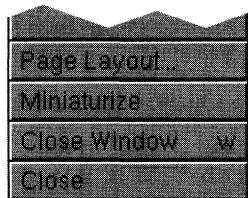


Figure 2-25. Close Commands

The Save, Save As, Save To, and Revert to Saved commands affect the main window and the document it displays. The Miniaturize and Close commands are alternatives to the title bar buttons with the same names; they can affect a window only if it has the corresponding button in its title bar.

An application can add its own commands to the Window menu immediately above the Page Layout command.

The Edit Menu

The Edit menu contains commands that alter the selection in the current key window (or in the main window if the key window doesn't respond to the command). A command is dimmed when it can't operate on the current selection.

Figure 2-26 illustrates the standard format of the Edit menu.

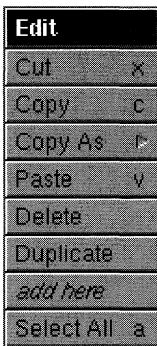


Figure 2-26. The Edit Menu

The commands shown in Figure 2-26 are described in the table below.

Command	Action
Cut	Deletes the current selection and copies it to the pasteboard.
Copy	Copies the current selection to the pasteboard without deleting it.
Copy As	Attaches a submenu that permits the user to copy the current selection to the pasteboard as a specified data type. The submenu lists the possible data types, as illustrated in Figure 2-27.
Paste	Replaces the current selection with the contents of the pasteboard.
Delete	Deletes the current selection without copying it to the pasteboard. The Delete key has the same effect.
Undo	Undoes the last editing change. This usually means all changes since the user last made a selection, including the selection of an insertion point.
Duplicate	Duplicates the current selection in an appropriate place and makes the duplicate the current selection. The “appropriate place” depends on the application. Graphics programs generally offset each copy a small amount from the original; a text editor might place the duplicate text on a new line.
Select All	Makes the entire contents of the file the current selection.

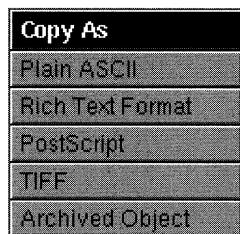


Figure 2-27. Copy As

Applications that permit the user to edit text or graphics should support at least the Cut, Copy, Paste, and Select All commands. Additional commands can be added to the Edit menu immediately above Select All.

The Font Menu

Applications that support text entry and editing should provide a Font menu and Font panel. The panel is defined in the Application Kit; it contains controls that set and preview fonts. The menu has a command to bring up the panel, and commands to make common adjustments to a font. It's illustrated in Figure 2-28.

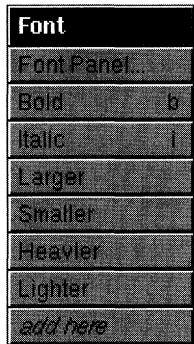


Figure 2-28. The Font Menu

Font menu commands are explained in the chart below.

Command	Action
Font Panel...	Brings up the Font panel and makes it the key window.
Bold	Makes the current selection bold, if it's not bold already, and makes it unbold if it is. The name of the command alternates between "Bold" and "Unbold" depending on the selection.
Italic	Makes the current selection italic or "oblique," if it isn't already, and makes it unitalic if it is. The name of the command alternates between "Italic" and "Unitalic" depending on the selection.
Larger	Makes the current selection one point larger.
Smaller	Makes the current selection one point smaller.
Heavier	Uses a heavier typeface to display the current selection.
Lighter	Uses a lighter typeface to display the current selection.

If the current selection is an insertion point, these commands affect the next set of characters inserted, rather than any existing text.

Each command leaves the other font attributes intact. For example, Bold will change 11-point Times® Roman to 11-point Times Bold and 24-point Courier Oblique to 24-point Courier Bold Oblique.

If there's more than one font in the selection, Larger and Smaller change each to be one point larger or smaller than its current size. The other commands make only the change that's appropriate for the first character in the selection. For example, if the first character in a multifont selection is italic, the (Un)italic command will remove the italic trait from all the text in the selection, but won't change any text that isn't italic. If the first character isn't italic, the same command (but now called "Italic") will italicize the entire selection, but won't alter any text that's already italic.

The Find Menu

Applications that display large amounts of text are encouraged to include a Find menu like the one illustrated in Figure 2-29. Other applications may also find this menu useful, but because it's designed most specifically for text, a variation of it may better meet their needs.

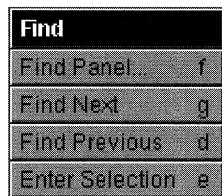


Figure 2-29. The Find Menu

Command	Action
Find Panel...	Brings up the Find panel, makes it the key window, and selects everything in its Find field so that the user can easily enter new text. If the panel is already on-screen, the command brings it to the front, makes it the key window, and selects the Find field.
Find Next	Searches forwards for the next occurrence of the string in the panel's Find field.
Find Previous	Searches backwards for the previous occurrence of the string in the panel's Find field.
Enter Selection	Enters the current selection into the panel's Find field so that Find Next and Find Previous can search for it.

Find Next and Find Previous begin searching at the current selection. If the search is successful, the text that's found is selected and becomes the starting point for the subsequent search. Neither command requires the Find panel to be on-screen. However, if the panel's Find field is empty, Find Next and Find Previous both bring up the Find panel, make it the key window, and select its Find field. This is exactly what the Find Panel command does. These other commands do it as convenience to the user, who has indicated an intention to do a search.

The Request Menu

Applications are encouraged to let users avail themselves of services provided by other applications, and to make their own services similarly available. One way that this is done is through the Request menu. Each command in the menu names both an action and an application to perform that action. Some examples are given in Figure 2-30.

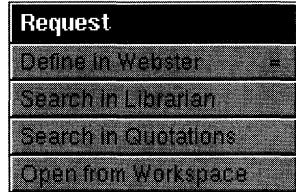


Figure 2-30. The Request Menu

Define in Webster passes the current selection to the Digital Webster application, which looks it up in the dictionary and thesaurus. Search in Librarian and Search in Quotations request those applications to search for a selected text string in the digital library. Open from Workspace sends a selected pathname to Workspace Manager, which asks the appropriate application to open the file.

Request commands conditionally activate the other application, but only if user input might be required. (Webster is likely to require the user to scroll the display, but Workspace Manager doesn't need the user's help to get a file opened. The application that opens the file will become active, but Workspace Manager won't.)

Each application that accepts a request from another application should publicize both the message it responds to and the command that can be used to send the message. It's important that the command be consistent across all applications that use it. The four commands shown in Figure 2-30 are standard for four applications they name—Webster, Librarian, Quotations, and Workspace Manager. You're encouraged to make use of these commands in your application and to define commands that other applications can use to send requests to your application.

Panels

Panels support the work done in the principal windows of an application. Like menus, most panels are vehicles through which the user can give instructions to the application. But unlike menus, they aren't restricted to a single column of commands; a panel can provide the user with a variety of different control objects—buttons, sliders, text fields, and more—arranged as best suits its purpose. The Font, Find, Page Layout, and Open panels are all examples. Such panels can be viewed as generalized and more versatile menus.

Some panels play a different role, however. Instead of letting the user give instructions to the application, they give information to the user. Help panels, the information panel, and attention panels that display warnings are examples.

What unites all panels—whether they convey instructions from the user to the application or information from the application to the user—is that they play conventional, supporting roles. None of them are sites for the user’s main work in the application. In a panel, the dialog between the application and the user is highly structured in both form and content.

Given their functional similarity to menus, it’s not surprising that panels share some menu attributes:

- Panels aren’t destroyed when they’re closed; they can again be brought to the screen.
- Panels generally aren’t miniaturized.
- The controls in a panel can respond to keyboard alternatives, even when the panel isn’t on-screen.
- A panel can never become the main window.

However, panels differ from menus in some of the same ways that standard windows do. Panels and standard windows share the same border and title bar styles, for example, and either can become the key window.

Attention panels differ from both standard windows and other panels in a number of ways. They have a distinctive look of their own and occupy one of the frontmost tiers on-screen.

Attention Panels

An attention panel demands attention from users by denying them the ability to work in any other window of the active application. Until it’s explicitly dismissed, the panel limits what the user can do within the application to just rearranging windows. Nothing else—title bar buttons, text entry, miniwindows, or controls in other panels—will work. The only menu commands that work are those that can affect the panel itself—for example, Cut, Copy, and Paste, if the panel includes a text field.

It’s possible to activate another application while an attention panel is on-screen, but when the user returns to the previous application, the mode created by the attention panel will still be in effect.

Types of Attention Panel

Attention panels are appropriate only in a limited number of situations. Because they create a mode that severely limits the user's freedom of action, their use should be restricted as much as possible. A panel can be made an attention panel when:

- It gives the user information about the current context. Such panels usually warn of an error, of a potentially dangerous or unexpected result of the user's current course of action, or of a condition that makes it impossible to carry out a requested action. But they may also simply supply information the user will need to proceed intelligently with the application.
- It interrupts an action to give the user an opportunity to take corrective steps—as, for example, the panel that interrupts the Quit command to let users save altered files before the application terminates.
- It clarifies or completes a user action—as, for example, the panel that completes the Save As and Save To commands.

Attention panels that interrupt or complete an action always give the user the option of canceling the action, in which case it's as if the user had never initiated the action in the first place. Panels that inform or warn should, if possible, let the user choose what to do in response to the information they convey.

Attention Panel Appearance

Because an attention panel sets an exclusive mode for itself, in effect disabling the rest of the application, it must be unmistakable and immediately apparent to the user. Some of the features that distinguish attention panels from other windows are illustrated in Figure 2-31. These features include:

- An empty title bar. The panel is labeled by text within its content area and is dismissed, not by a close button, but by buttons within the content area.
- Larger than usual type.
- An icon to identify its application.
- A Cancel button that lets the user cancel the action that brought the panel up.
- The Return symbol,  , on a button with the panel's default action. In Figure 2-31, it's the Quit button, but it could also have been the Cancel button.

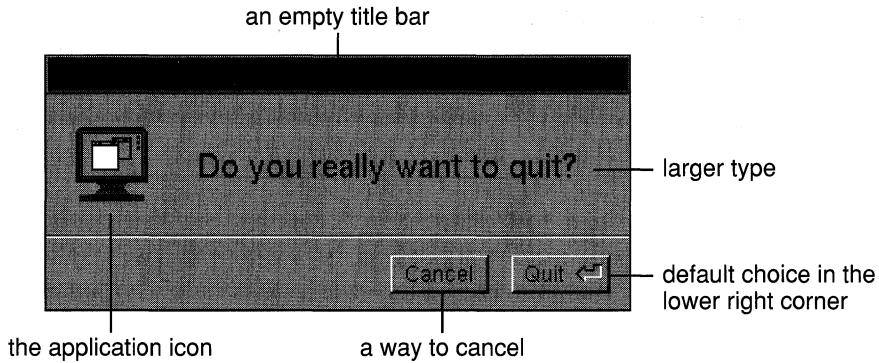


Figure 2-31. Attention Panel

Attention Panel Behavior

It's not enough that attention panels look different than other windows; they must also behave differently, in ways that keep the user's attention. For this reason, an attention panel is always the key window (when its application is active) and stays in front of all other windows on the screen (except pop-up and pull-down lists the user activates from the panel itself). Applications should place their attention panels squarely in front of the user at the center of the screen.

Unlike other panels, an attention panel stays on-screen when the user chooses another active application. The panel can't be buried or forgotten; the user may need to move it to the side to keep it from obscuring the windows of the other application.

When the user returns to the application that put up the attention panel, the panel again becomes the key window—even if the user activated the application by clicking in a different window.

Sometimes, an application other than the active application may put an attention panel on-screen. For example, after the user prints a WriteNow file, the application handling the print job could bring up a panel with a “printer out of paper” warning. Since this attention panel doesn't belong to the active application, it won't be the key window. It therefore won't have a black title bar and the Return key won't work to activate its default button.

Dismissing an attention panel by clicking one of its buttons activates its application, just as any other click within the window would.

Dismissing an Attention Panel

Attention panels are dismissed from the screen as soon as the user takes the required action, which can be as simple as typing Return. When dismissed, the panel's mode ends.

Even attention panels that give warnings require user action to dismiss them and end the mode. This forces the user to take note of the warning.

Each action that can dismiss an attention panel is represented by a separate button inside its content area. In contrast, an ordinary panel is closed only by its close button (or the Close command), never by a button in the content area.

The buttons that dismiss an attention panel should be located along the right and lower edges of the panel, with the default button—the one operated by Return—in the lower right corner. The Return key is used as a shortcut for the action the user is most likely to take, provided that action is also not destructive. Actions the user might regret—such as deleting a file or removing recent editing changes—shouldn't be made easier with the Return shortcut.

Each button must be clearly labeled with the action it performs. Generic labels (like “Yes,” “No,” and “OK”) are not permitted, since they require the user to look elsewhere, perhaps to other text, to understand what the button actually does. However, a button labeled “OK” can be used to dismiss an attention panel that issues a warning.

If an attention panel completes or interrupts a user action, one of its buttons must be labeled “Cancel”.

Control Panels

Panels that aren't attention panels—“control panels”—can be left on-screen as the user works in the application. They're brought to the screen by menu commands, just as submenus are. But they don't attach to a supermenu and remain at hand until they're closed.

Control panels differ from attention panels in a number of ways:

- They aren't isolated into a tier of their own above other windows; they compete with standard windows for space on the screen.
- They generally aren't visible unless they belong to the active application; they rarely persist on-screen once the application has been deactivated.
- They generally become the key window only if they accept characters from the keyboard.

- They're closed by a close button in the title bar, rather than by a button in the content area.
- They display a title in the title bar like standard windows.

Persisting Panels

By default, a control panel is removed from the screen when its application is deactivated. The user sees only panels related to the active application. This prevents confusion—such as might arise when there are similar Find panels for two different applications on-screen at once.

An application can override this default behavior and allow its panels to remain on-screen after it has been deactivated, but only if they contain information that would be pertinent to the user's activities in another application. This should be a rare occurrence.

Relinquishing Key Window Status

A control panel should remain the key window only as long as necessary. If user actions within the panel affect the main window, key window status should be returned to the main window as soon as those actions are completed.

For example, when the user clicks the Set button (or types Return) in a Font panel to change the font of the current selection in the main window, the panel gives up being the key window. In all likelihood, the user is finished with the Font panel (at least until the selection has changed) and is ready to resume working in the main window. Under these circumstances, the user should be free to begin working in the main window immediately, without being forced to click it once just to make it the key window.

The Information Panel

The panel controlled by the Info command in the main menu provides the user with basic information about the application. Although each application can decide what type of information to provide, it's recommended that it minimally include:

- The name of the application
- The application icon
- Copyright information
- The current version of the application

The information panel should not be implemented as an attention panel.

Controls

Controls are graphical objects that users manipulate with the keyboard and mouse to give instructions to an application. They’re patterned after familiar control devices from everyday life—switches, knobs, forms, gauges, and the like—and perform analogous functions. Like the dials and levers on a machine, graphical control objects let the user “operate” an application.

Every control responds visually to direct manipulation by the user—a dial turns, a button pushes in or highlights, the knob of a slider slides. Controls go beyond this direct response, however, to cause the application to do something. They, in effect, translate the user’s direct manipulation into an instruction for the application. A button sets a state or initiates a program action, a slider sets a value, a menu item sends a command, and so on.

Which keyboard and mouse actions a control responds to and how it reacts visually are part of the definition of the control; they’re discussed further in this section. What the control causes an application to do is part of the definition of the application; it depends solely on how the application uses the control. In this respect, graphical controls are no different from control devices in the real world. For example, identical mass-produced switches can be installed on a variety of different machines. The manufacturer of the switch provides it with a user interface; the installer gives it specific meaning for a specific machine. (See Chapter 7 for information on how to “install” the controls defined for you in the Application Kit.)

The Application Kit defines five canonical controls:

- Sliders
- Buttons
- Menu commands
- Text fields
- Scrollers

Because they’re widely used, each of these controls is described in some detail in its own section. Menu commands were described under “Menus” above; the others are described in the sections below.

You can also design your own controls—the Application Kit makes this relatively easy—but they should adhere to these basic design principles:

- Every control must provide immediate feedback to let the user know that an action has “taken.” Just as users can look at a dial on a stove to see whether it has been turned, a graphical control must alter its appearance in response to user actions. It shouldn’t depend on a reaction elsewhere in the application to give the user feedback.
- Every control should have a distinctive appearance and behavior. Don’t design controls that look so similar to the canonical controls that users will confuse one with the other.

- The behavior of a control should be apparent from its appearance. After a bit of familiarity with the NeXT computer, users should be able to easily recognize a control object and know almost instinctively how to operate it.

Sliders

A slider is a device that sets a value. As illustrated in Figure 2-32, it consists of a vertical or horizontal *bar* and a *knob* that moves on the bar.

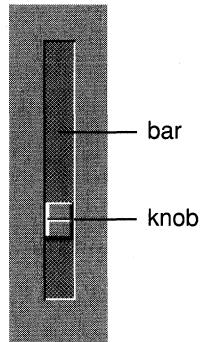


Figure 2-32. Slider

The position of the knob in the slider indicates its current value. Users can move the knob, and thus alter the value, by positioning the cursor anywhere over the bar (even the part of the bar that's covered by the knob) and pressing the left mouse button. The knob immediately jumps to the location of the cursor. The user can release the mouse button to fix the knob in its new location, or begin dragging the knob along the bar.

A slider can set values on a continuous scale (between some maximum and minimum) or values at discrete intervals. If the latter, the knob jumps to the position of the nearest permitted value when the user releases the mouse button.

Buttons

Buttons are the primary controls for setting a state or initiating an application action. They're used for the controls in title bars (the miniaturize and close buttons), for Cancel and the other choices that dismiss attention panels, and in most other situations where a basic control device is called for.

Buttons can assume a variety of different shapes and sizes, making it more appropriate sometimes to refer to them as "switches," "check boxes," or "toggles," rather than as "buttons." Some of the variety is illustrated in Figure 2-33.

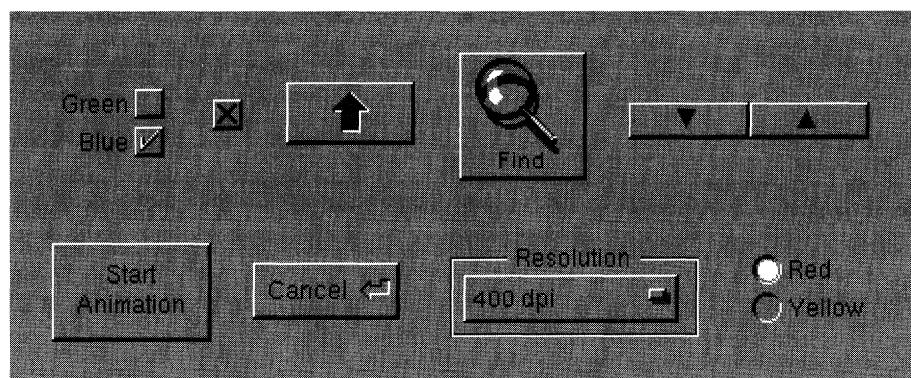


Figure 2-33. An Assortment of Buttons

All buttons respond to a click; some also respond to being pressed. Those that respond to being pressed send an instruction to the application immediately as the user pushes the mouse button down. Typically, they repeat the instruction at regular intervals—as long as the mouse button is held down and the cursor is kept over the button on-screen—for a continuous, iterative action. Users can drag away from the button and back again to stop and restart the action. A button that responds only to being clicked sends its instruction to the application when the user releases the mouse button, provided the cursor is over the button on-screen.

Whether it responds to being clicked or to being pressed, a button changes its appearance as soon as the mouse button goes down. It retains its altered appearance while it's under the cursor and the mouse button remains down. When the user releases the mouse button, the button on-screen keeps its altered appearance long enough for its instruction to be carried out. Usually this is momentary (though it need not be), so users generally notice the button changing as soon as the click is over.

A button's appearance during a click (or while it's pressed) can change in any of four ways:

- It can highlight.
- It can appear to be pushed in.
- It can both highlight and appear to be pushed in.
- If the button displays a bitmap image (as title bar buttons do), it can alter the bitmap it displays.

These changes are illustrated for buttons with just one state in Figure 2-34.

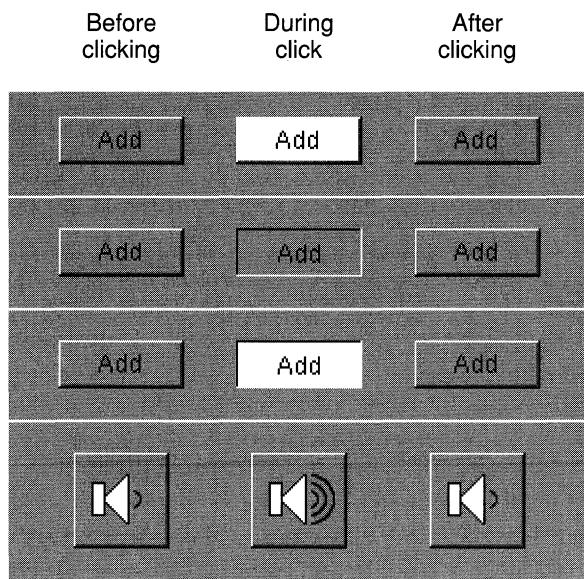


Figure 2-34. One-State Buttons

Buttons that are used to set a state display one state before being clicked and another after the click. The difference in state is generally shown by the presence or absence of highlighting or by changing the icon the button displays. It can also be accomplished by changing the button's label. The possibilities are illustrated in Figure 2-35.

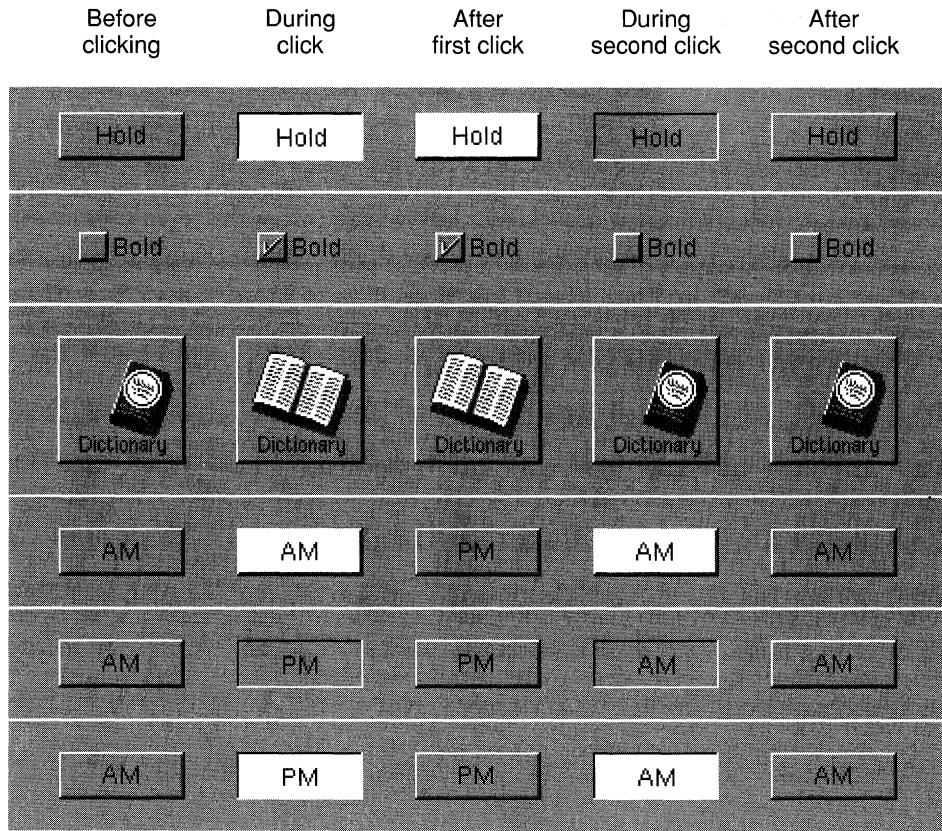


Figure 2-35. Two-State Buttons

Figure 2-35 also illustrates some of the principles that determine how a button looks during a click:

- A button must change its appearance during a click, as soon as the mouse button goes down.
- The appearance of a button during a click should reflect what's about to happen. Buttons that display a state should reflect the state after the click.
- If highlighting is used to indicate state, it shouldn't also be used to give feedback during a click. Some other change (such as the appearance of being pushed in) should be used instead.

A button's label usually states, in a succinct shorthand, what action it causes the application to take. Even when a button purports to label a state (as in Figure 2-35), users are apt to think of it not as the current state, but as the state that will be set if the button is clicked. In other words, they're liable to interpret it as an action. An "On" button, for example, is more likely to be interpreted to mean "Press this to turn something on," than "This is now on."

It's best, therefore, to use icons and highlighting to show the current state, and reserve the button's label as a brief statement of what the button does. Buttons that do label a state, like the "AM" and "PM" buttons in Figure 2-35, should be used only where what they label is clearly visible. The buttons in Figure 2-35 could be used alongside a digital representation of the time, but they can't stand alone.

A button with an action label shouldn't change the action it performs. Although it's sometimes tempting to alter the action with the button's state—to switch between "Start" and "Stop," or between "Erase" and "Restore," for example—it's best to provide a different button for each action and disable those that aren't operable. Where necessary, a pop-up list can be used to save screen space.

Button labels should be capitalized like menu commands: The first and last words begin with uppercase letters and the words between are capitalized as they would be in a title.

Dimming the label of a button (using gray text) indicates that the button is disabled.

Text Fields

A text field is a slot where the user can type in a single line of data—such as a file name, a part number, or an address. The text is editable and selectable; the data is entered only when the user types Return or clicks a button that's associated with the field.

Text fields can be titled and arranged in groups to produce an on-screen form, such as the one illustrated in Figure 2-36.

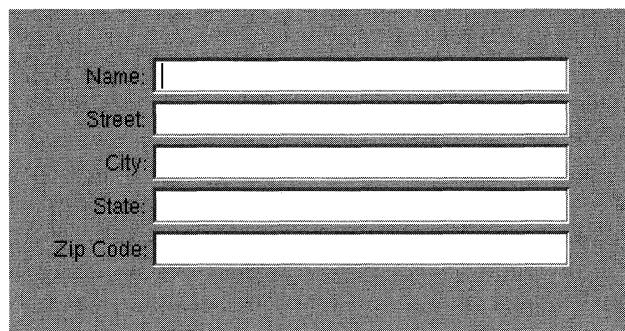


Figure 2-36. Text Fields in a Form

Like all editable text, a text field has a white background when the user is entering data, and a light gray background when the entire field is selected. To indicate that it's editable, a text field is surrounded by a bezeled border that makes it appear inset from the surface of the screen. Figure 2-36 illustrates this border.

When there's more than one text field in a window, the Tab key can move the selection—the point where typing will appear—from one field to another:

Tab	Moves from one text field to the next one in series. For example, in the form illustrated in Figure 2-36, Tab would cause the current selection to jump from the name field to the street address field to the city field, and so on.
Shift-Tab	Moves from one text field to the previous one in series.

When the user presses the Return key after typing in a text field, the field makes something happen. Data might be entered and processed, a search might begin for text that matches the string in the field, or a document might be saved to a file name the user typed. Exactly what happens is up to the application. To let users know what to expect, it's recommended that you include a button in the display to act as the equivalent of Return. The button's label is an explicit reminder of what Return will do; from the user's point of view, Return is simply a shortcut for the action of the button.

There's also direct feedback that Return has taken effect:

- If the text field is in an attention panel, Return may dismiss the panel.
- If the text field is one in a series, Return may select the next field (just as Tab would).
- If the action of the text field is repeatable, Return may select all the text in the same field so the user can easily replace it.
- If a button is associated with the text field, Return may cause the button to act as if it had been clicked.

In some cases, where a text field is part of a form, Return may not perform any particular action of its own. Instead, it will do just what Tab does—move the selection to the next field. Action on a button or other control is required to enter data typed into the form.

Generally, text fields accept unrestricted data, but sometimes an entry won't be acceptable if it's the wrong data type—if, for example, the user types in a floating-point number when an integer is called for. Text fields generally recognize these five restricted types of data:

- Unsigned integers
- Signed integers
- Unsigned floating-point numbers
- Signed floating-point numbers
- Dates

If the user's entry isn't acceptable, all of the text in the field is selected and highlighted. The user can make any necessary corrections and try again.

If the user enters more text than will fit in the field, the entry is automatically scrolled so that the insertion point stays visible.

Scrollers

Scrollers are used to control what's displayed within a window, or within a rectangular subsection of a window. When the material to be displayed is larger than the opening available to display it, the user must scroll unseen portions into the opening in order to view them. Figure 2-37 shows, diagrammatically, a scrollable document, the area available to view it, and the scrollers that can move the opening around on the surface of the document.

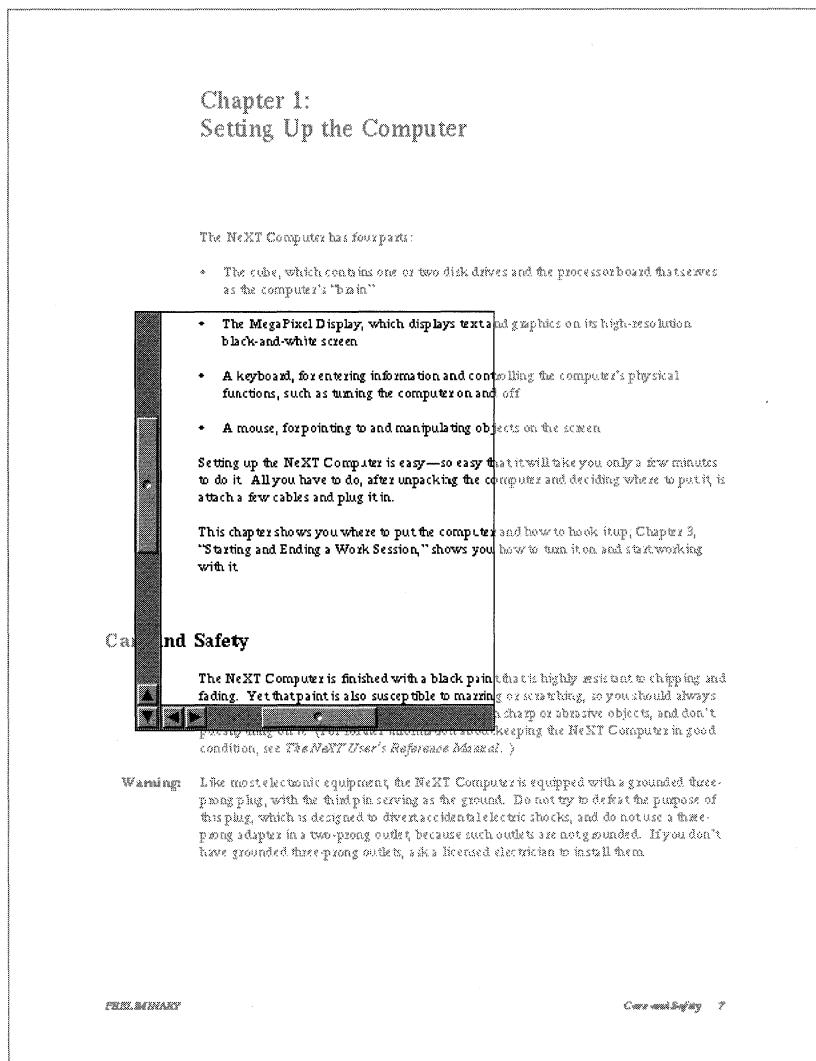


Figure 2-37. Scrollable Document

As illustrated in Figure 2-37, a scroller has just three parts, a *bar*, a *knob*, and an optional set of *scroll buttons*. Figure 2-38 shows a vertical scroller, which scrolls a document up and down. A horizontal scroller scrolls from side to side.

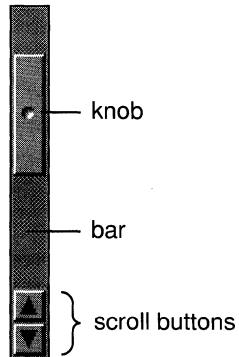


Figure 2-38. Vertical Scroller

Scroller Layout

If a document is taller than the opening available to view it, it's provided with a vertical scroller. If it's wider than the opening, it's provided with a horizontal scroller. The layout of both scrollers was illustrated in Figure 2-37 above.

The horizontal scroller is located along the bottom edge of the opening, so it won't be adjacent to the window's title bar. The vertical scroller is positioned along the left edge of the opening. Since text is generally aligned at the left margin, this keeps the scroller close to the display it controls. The user can drag a window partially off-screen (or under the dock) to the right and still scroll a meaningful amount of text into view.

The scroll buttons for both vertical and horizontal scrollers occupy the lower left corner, where the two scrollers meet. Keeping all the scroll buttons in the same region makes it easy for users to move from one set to the other.

If a document is provided with scrollers, but happens, perhaps temporarily, to fit within the opening, one of two things happen:

- The scrollers disappear and light gray strips appear in their place. The strips indicate that the document will be scrollable, should it grow larger than the opening.
- The scrollers remain, but indicate that the document isn't scrollable because the knob fills the bar.

The Knob and Bar

The bar of a scroller represents the entire scrollable document; the knob represents the part of the document that's visible. The placement of the knob in the bar shows which part is currently visible in the opening. On a vertical scroller, the height of the knob relative to the height of the bar indicates how much of the document, from top to bottom, is visible. On a horizontal scroller, the width of the knob indicates how much of the document is visible from side to side. The knob shrinks as the user adds material to the document, and grows as material is deleted. However, the knob never shrinks to be smaller than a square.

Users scroll the display by moving the knob in the bar. The knob can be moved in four ways:

- By dragging it to a new location. The display is adjusted as the knob moves.
- By clicking in the bar (outside the knob). The knob jumps to the location of the click, and the display is adjusted accordingly. If the user doesn't immediately release the mouse button from the click, the knob can be dragged to a new location. This permits users, in a single mouse action, to select the general part of a document they want to view (by clicking in the bar) and then to adjust the display (by dragging the knob).
- By clicking or pressing the scroll buttons. The arrows on the scroll buttons point in the direction the knob will move.
- By extending a selection outside the opening where it's displayed. This automatically scrolls unseen portions of the selection into view.

By moving the knob in the bar, users metaphorically move an opening around on the surface of the document so that they can see the portions they desire. Visually, of course, it's the document that appears to move, not the opening. This means that the knob and the display move in opposite directions. To avoid confusion, documentation should concentrate on the metaphor of adjusting the portion of the document that's visible, rather than adjusting the document to make it visible.

The Scroll Buttons

The scroll buttons permit more precise scrolling than direct manipulation of the knob. When clicked, a vertical scroll button scrolls a single line of text. When pressed, it repeatedly scrolls one line after another. If the document displays something other than text (graphics perhaps), the application can determine the precise distance to scroll; it's always a distance comparable to a single line of text. Horizontal scroll buttons work in a similar way, scrolling a small amount in a horizontal direction.

The two scroll buttons on the same scroller form a related pair. When the user drags from one to the other without releasing the mouse button, each button acts as if it had been pressed. It's not possible to slide from the scroll buttons on one scroller to those on the other scroller, however.

When the Alternate key is held down, the scroll buttons scroll one viewful at a time. Generally, when scrolling down a document, the bottom line (or two) is redisplayed at the top of the opening each time the display changes. When scrolling toward the beginning of a document, the top line (or two) is redisplayed at the bottom. This provides users with a bit of overlapping context and reassures them that nothing was skipped over when the display changed.

Sometimes scroll buttons appear alone, without the rest of the scroller—for example, in Workspace Manager’s Directory Browser. Since the knob and bar aren’t present to indicate when it’s impossible to scroll further in one direction or the other, the arrow on the scroll button is dimmed when the button won’t work.

Automatic Scrolling

When the user begins a selection in the visible part of a document then drags outside the opening, the document will scroll continuously to bring more of the selection into view, until the user releases the mouse button. The farther the user drags outside the opening, the greater each repeated change in the display. It’s as if the application tries repeatedly to bring the point under the cursor into view.

As the document scrolls, the scroller knob is adjusted to reflect the current position of the display.

Fine Tuning

If a document is large, small movements of the knob may correspond to sweeping changes in the display. This makes it difficult for users to adjust the display with precision when dragging the knob.

To make fine adjustments possible even for large documents, some scrollers have a “fine tuning” mode. While the Alternate key is held down, the knob and display move only slightly in response to large movements of the mouse. In this mode, the knob moves in the direction it’s dragged, but doesn’t stay with the cursor; it continues to reflect the position of the document being displayed.

Once the Alternate key is released, any subsequent dragging action will cause the knob to jump to the position of the cursor.

Chapter 3

Object-Oriented Programming and Objective-C

3-3 Objects

3-4 Messages

- 3-6 Messages and Function Calls
- 3-6 Dynamic Binding

3-7 Classes

- 3-8 Inheritance
 - 3-9 Inheriting Instance Variables
 - 3-9 Inheriting Methods
 - 3-10 Overriding One Method with Another
 - 3-10 Abstract Superclasses
- 3-10 Class Definitions
 - 3-11 The Interface
 - 3-12 Separating the Interface from the Implementation
 - 3-12 Importing the Interface
 - 3-13 The Implementation
 - 3-15 Adding to a Class
 - 3-16 Variables and Class Objects

3-16 How Messaging Works

- 3-19 Selectors
 - 3-19 Varying the Message at Run Time
 - 3-20 Identifying Return and Argument Types
- 3-20 Hidden Arguments
- 3-21 Messages to **self** and **super**
 - 3-22 An Example
 - 3-23 Using **super**
 - 3-24 Redefining **self**

3-25 The Object Class

- 3-26 Memory Management
- 3-26 Class Initialization
- 3-26 Avoiding Messaging Errors
- 3-27 Archiving Support
- 3-28 Inheritance Relationships
- 3-29 Posing

3-29	Options
3-30	Static Typing
3-31	Public Instance Variables
3-32	Return and Argument Types
3-33	Getting a Method Address
3-33	Getting an Object Data Structure

3-34 Type Encoding

3-36	Language Synopsis
3-36	Messages
3-36	Defined Types
3-37	Preprocessor Directives
3-37	Compiler Directives
3-37	Method Declarations
3-38	Method Implementations
3-38	Other Keywords

Chapter 3

Object-Oriented Programming and Objective-C

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/ReleaseNotes/ObjC.rtf](#)
[/NextLibrary/Documentation/NextDev/ReleaseNotes/AllocInitAndNew.rtf](#)
[/NextLibrary/Documentation/NextDev/ReleaseNotes/Zones.rtf](#)

This chapter discusses the principles of object-oriented programming as they're implemented in the Objective-C language, the language used in the NeXT software kits. Programs based on the kits must be written in Objective-C.

Objective-C is an extension of the C language; its syntax is a superset of standard C syntax, and its compiler works for both C and Objective-C source code. The compiler recognizes Objective-C source files by a ".m" extension, just as it recognizes files with only standard C syntax by a ".c" extension. As implemented by NeXT, the Objective-C language is fully compatible with ANSI standard C.

Because object-oriented programs postpone many decisions from compile time to run time, the Objective-C language depends upon a run-time system for executing the compiled code. This discussion presents the language—and important elements of the run-time system—as they're implemented for the NeXT computer. NeXT has modified the GNU C compiler to also compile Objective-C and provides its own run-time system.

Throughout this manual and in other NeXT documentation, the term "Objective-C" refers to the language as presented here.

You can find more extensive discussions of object-oriented programming in any of several books that have been published on the subject. Some titles are listed under "Suggested Reading" in the *NeXT Technical Summaries* manual. A formal grammar of the Objective-C extensions to C is also presented in *Technical Summaries*.

Objects

As the name implies, object-oriented programs are built around *objects*. An object associates data with the particular operations that can use or affect that data. These operations are known as the object's *methods*; the data they affect are its *instance variables*. In essence, an object bundles a data structure (instance variables) and a group of procedures (methods) into a self-contained programming unit.

For example, through the Application Kit, you can produce an object that displays a matrix of cells to users of your application. The cells might be text fields where the user can enter data, a series of mutually exclusive switches, a list of menu commands, or a bank of sliders. A Matrix object contains instance variables that define the matrix, including its dimensions and coordinates, the font used to display character strings in the cells, the arrangement of cells into rows and columns, and the cells themselves. The Matrix can apply methods that do such things as alter its size, change its position on-screen, add and remove cells, set the color that's displayed between cells, and highlight the user's selection.

An object's instance variables are private to the object; you get access to them only through the object itself. Moreover, an object sees only the methods that were designed for it; it can't mistakenly perform methods intended for other types of objects. This encourages a style of structured programming that isolates each problem a program must solve into a separate object. Just as a C function localizes its automatic variables, hiding them from the rest of the program, an object hides both its instance variables and its methods.

In Objective-C, objects are identified by a distinct data type, **id**. This type is defined as a pointer to an object (in reality, a pointer to the object's data structure). Like a C function or an array, an object is identified by its address. All objects, regardless of their instance variables or methods, are of type **id**. For the object-oriented constructs of Objective-C, **id** replaces **int** as the default data type. (For strictly C constructs, such as function return values, **int** remains the default type.)

Note: Objects can also be more restrictively typed, based on their particular data structures. (See "Classes" and "Options" below for details.)

The keyword **nil** is defined as a null object, an **id** with a value of 0. **id**, **nil**, and the other basic types of Objective-C are defined in the header file **objc.h**, which must be included in every Objective-C program. It's in the **objc** subdirectory of **/usr/include**.

Messages

To get an object to do something, you must send it a *message* telling it to apply a method. In Objective-C, *message expressions* are enclosed in square brackets:

[*receiver message*]

The receiver is an object, and the message tells it what to do. In source code, the message is simply the name of a method and any arguments that are passed to it. When a message is sent, the run-time system selects the appropriate method from the receiver's repertoire and has the receiver apply it. For this reason, the method name in a message is called a method *selector*.

For example, this message tells the **myMatrix** object to perform its **display** method, which draws the matrix and its cells in a window:

```
[myMatrix display];
```

Methods can also take arguments. The message below tells **myMatrix** to change its location within the window to coordinates (30.0, 50.0):

```
[myMatrix moveTo:30.0 :50.0];
```

Here the method name (selector), **moveTo::**, has two colons, one for each of its arguments. The arguments are inserted after the colons, breaking the name apart. Colons don't have to be grouped at the end of a method name, as they are here. Usually a keyword describing the argument precedes each colon. The **getRow:andCol:ofCell:** method, for example, takes three arguments:

```
int row, column;  
[myMatrix getRow:&row andCol:&column ofCell:someCell];
```

This message finds the location of **someCell** in the matrix and puts the row and column where it's located in the two variables provided.

Methods that take a variable number of arguments are also possible. Extra arguments are separated by commas after the end of the method name. (Unlike colons, the commas aren't considered part of the name.) In the following example, the imaginary **makeGroup:** method is passed one required argument (**group**) and three that are optional:

```
[receiver makeGroup:group, memberOne, memberTwo, memberThree];
```

Like standard C functions, methods can return values. The following example assigns the gray value returned by the **backgroundGray** method to the variable **shade**.

```
float shade;  
shade = [myMatrix backgroundGray];
```

A message to **nil** also is valid,

```
[nil moveTo:100.0 :22.5];
```

but it has no effect and makes little sense. Messages to **nil** simply return **nil**.

Messages and Function Calls

As the examples above illustrate, messages in Objective-C can appear in the same syntactic positions as function calls in standard C. But, because methods “belong to” an object, messages behave differently than function calls:

- An object has access only to the methods that it can perform. It can’t confuse them with the methods of another object, even if the other object has a method with the same name. This means that two objects can respond differently to the same message. For example, each object sent a **display** message could display itself in a unique way. This feature, which plays a significant role in the design of object-oriented programs, is sometimes referred to as *polymorphism*.
- A method has automatic access to all the receiving object’s instance variables. You don’t need to pass them to the method as arguments.

A method has access only to the receiver’s instance variables. If it requires information about a variable stored in another object, it must send a message to the object asking it to reveal the contents of the variable. The **backgroundGray** method in the example above is used for just this purpose. It returns the value stored in one of **myMatrix**’s instance variables.

Dynamic Binding

A crucial difference between function calls and messages is that a function and its arguments are bound together in the compiled code, but a method and a receiving object aren’t united until the program is running and the message is sent. A method is “called” through a run-time messaging routine that locates the method named by the selector and passes the object’s instance variables to it. (For more on this routine, see “How Messaging Works,” below.)

This *dynamic binding* of methods and receivers works hand-in-hand with polymorphism to give object-oriented programming much of its flexibility and power. Since each object can have its own version of a method, a program can achieve a variety of results, not by varying the message itself, but by varying just the object that receives the message.

This can be done as the program runs. Since messages aren’t bound to receivers until run time, receivers can be decided “on the fly” and can be made dependent on user actions. In the Application Kit, for example, users determine which objects receive messages from menu commands like Cut, Copy, Paste, and Close.

Classes

An object-oriented program is typically built from a variety of objects. A program based on the NeXT software kits might use Matrix objects, Window objects, List objects, SoundView objects, Text objects, and others. Programs often use more than one object of the same kind or *class*—several Lists or Windows, for example.

In Objective-C, you define objects by defining their class. The class definition is a prototype for a kind of object; it declares the instance variables that become part of every object belonging to the class and defines a set of methods that all objects in the class can use.

The compiler creates just one visible object for each class, a *class object* that knows how to build new objects belonging to the class. (For this reason it's sometimes also called a “factory object.”) The class object is the compiled version of the class; the objects it builds are *instances* of the class. The objects that will do the main work of your program are instances created by the class object at run time.

Note: The compiler also builds a “metaclass object” for each class. It describes the class object just as the class object describes instances of the class. But while you can send messages to instances and to the class object, the metaclass object is used only internally by the run-time system.

This code tells the Matrix class object to create a new Matrix instance and assign it to the **myMatrix** variable:

```
id myMatrix;
myMatrix = [Matrix new];
```

This last line of code, or one like it, would be necessary before **myMatrix** could receive any of the messages that were illustrated in the previous examples. Every class object has at least one method (like **new**) that enables it to produce new objects. These methods often take arguments to initialize the new instance and have keywords to label the arguments (**newFrame:text:alignment:**, for example), but they all generally begin with “new”.

By convention, class names (here “Matrix”) begin with an uppercase letter; the names of instances (here “myMatrix”) typically begin with a lowercase letter. Unlike instance names, class names don’t identify variables. They can be used in only two, very different contexts:

- As the receiver in a message expression, the class name refers to the class object. This usage was illustrated in the example above. The class name can stand for the class object only as a message receiver. In any other context, you must ask the class object to reveal its **id** (by sending it a **class** message). The example below assigns the Matrix class object to the **myClass** variable:

```
id myClass;
myClass = [Matrix class];
```

- The class name can also be used as a type name for instances of the class. For example:

```
Matrix *anObject;
anObject = [Matrix new];
```

Here **anObject** is *statically typed* to be a Matrix. The compiler will expect it to have the data structure of a Matrix instance. Static typing enables the compiler to do better type checking and permits other optimizations. See “Options” later in this chapter for details.

Inheritance

All classes are linked together in a hierarchical tree with a single class, the Object class, at its root. Every class (but Object) has a *superclass* one step nearer the root, and any class (including Object) can be the superclass for any number of *subclasses* one step farther from the root. Figure 3-1 illustrates the hierarchy for a few of the classes in the NeXT Application Kit.

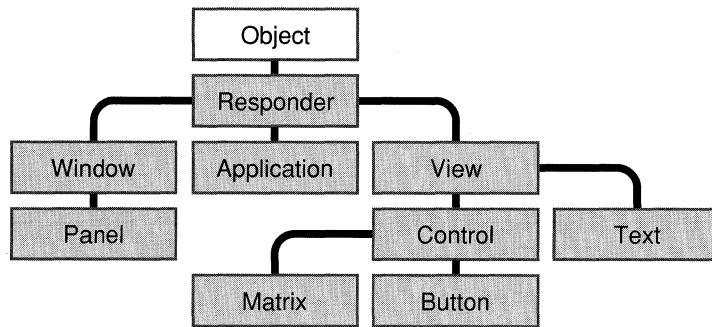


Figure 3-1. Some Application Kit Classes

When you define a class, you must link it to the hierarchy by declaring its superclass; every class you create must be the subclass of another class. Plenty of potential superclasses are available:

- The Object class and a handful of others are provided with the run-time system.
- The Application Kit defines a variety of classes for objects that can draw on the screen and respond to user actions on the keyboard and mouse.
- Sound Kit has classes that provide a high-level abstraction for recording, playing, visualizing, and editing sounds.
- Music Kit defines classes for music composition, synthesis, notation, and performance.

You can create new subclasses for any of these classes; you can also create subclasses for any of the classes you create.

Each class *inherits* both instance variables and methods from its superclass.

Inheriting Instance Variables

When a class object creates a new instance, the new object contains not only the instance variables that were defined for its class, but also the instance variables defined for its superclass, and for its superclass's superclass, all the way back to the root Object class. The instance variables for the Object class become part of every object.

Figure 3-1 above shows that the Matrix class is a subclass of the Control class; the Control class is a subclass of View; View is a subclass of Responder; and Responder is a subclass of Object. So a Matrix object includes the instance variables defined for Control, View, Responder, and Object, as well as those defined specifically for Matrix. This is simply to say that a Matrix object isn't only a Matrix, it's also a Control, a View, a Responder, and an Object.

For example, if a variable is statically typed to be a View, you could assign a Matrix instance to it:

```
View *myView;
myView = [Matrix new];
```

This is possible because a Matrix is a View. It's more than a View since it also has the instance variables of a Control and a Matrix, but it's a View nonetheless.

Inheriting Methods

An object has access not only to the methods that were defined for its class, but also to methods defined for its superclass, and for its superclass's superclass, all the way back to the root of the hierarchy. A Matrix object can use methods defined in the Control, View, Responder, and Object classes as well as methods defined in its own class.

Any new class you define in your program can therefore make use of the code written for all the classes above it in the hierarchy. This type of inheritance is a major benefit of object-oriented programming. When you use one of the object-oriented kits provided by NeXT, your programs can take advantage of all the basic functionality coded into the kit classes. You have to add only the code that customizes the kit to your application.

Class objects also inherit from the classes above them in the hierarchy. But because they don't have instance variables (only instances do), they inherit only methods.

Overriding One Method with Another

There's one useful exception to inheritance: When you define a new class, you can implement a new method with the same name as one defined in a class farther up the hierarchy. The new method overrides the original; instances of the new class will perform it rather than the original, and subclasses of the new class will inherit it rather than the original.

Although overriding a method blocks the original version from being inherited, other methods defined in the new class can sometimes skip over the redefined method and find the original (see “Messaging to **self** and **super**,” below, to learn how). The redefined method can also incorporate the very method it overrides. When it does, the new method serves only to refine or modify the method it overrides, rather than replace it outright.

Although a subclass can override inherited methods, it can't override inherited instance variables. Since an object has memory allocated for every instance variable it inherits, you shouldn't try to override an inherited variable by declaring a new one with the same name. If you do, errors will result.

Abstract Superclasses

Some classes are designed only so that other classes can inherit from them. These *abstract superclasses* group methods and instance variables that will be used by a number of different classes into a common definition. The Object class is the prime example of an abstract superclass. Although programs often define Object subclasses and use instances belonging to the subclasses, they never use instances belonging directly to the Object class.

Class Definitions

Much of object-oriented programming consists of defining new object classes. In Objective-C, classes are defined in two parts:

- A file that declares the interface to the new class
- A file that actually defines the class (contains the code that implements it)

Each class requires both files; the declaration of the interface and the class implementation can't be in the same file. However, a single file can declare more than one class, or implement more than one class. Nevertheless, it's customary to have separate interface and implementation files for each class.

Interface and implementation files typically are named after the class. The implementation file has a “.m” suffix, indicating that it contains Objective-C source code. The interface file can be assigned any other extension. Because it's included in other source files, the interface file usually has the “.h” suffix typical of header files. For example, the Matrix class is declared in **Matrix.h** and defined in **Matrix.m**.

The Interface

The declaration of a class interface begins with the compiler directive **@interface** and ends with the directive **@end**. (All Objective-C directives to the compiler begin with “@”.)

```
@interface ClassName : ItsSuperclass
{
    variableDeclarations
}
methodDeclarations
@end
```

The first line of the declaration presents the new class name and links it to its superclass. The superclass defines the position of the new class in the inheritance hierarchy, as discussed under “Inheritance” above.

Following the class declaration, braces enclose declarations of *instance variables*, the data structures that will be part of each instance of the class. Here’s a partial list of the instance variables declared in the Matrix class:

```
id      selectedCell;
int     numRows;
int     numCols;
float   backgroundGray;
id      cellClass;
```

Methods for the class are declared next, after the braces enclosing instance variables and before the end of the class declaration. The names of methods that can be used by class objects, *class methods*, are preceded by a plus sign:

```
+ new;
```

The methods that instances of a class (objects created by the class object) can use are called *instance methods*, and are marked with a minus sign:

```
- display;
```

Although it’s not a common practice, you can define a class method and an instance method with the same name.

Method return types are declared using the standard C syntax for casting one type to another:

```
- (float)backgroundGray;
```

Argument types are declared in the same way:

```
- setTag:(int)anInt;
```

If a return or argument type isn't explicitly declared, it's assumed to be the default type for methods and messages—an **id**. The **new**, **display**, and **setTag**: methods illustrated above all return **ids**.

When there's more than one argument, they're declared within the method name after the colons. They break the name apart in the declaration, just as in a message. For example:

```
- moveTo:(NXCoord)x :(NXCoord)y;  
- getRow:(int *)aRow andCol:(int *)aColumn ofCell:aCell;
```

Objective-C borrows this syntax from Smalltalk, one of the first object-oriented languages, rather than from standard C. (NXCoord is a defined type for floating-point values that specify coordinate measurements.)

Methods that take a variable number of arguments declare them just as a function would:

```
- makeGroup:group, ...;
```

Separating the Interface from the Implementation

The purpose of the interface file is to declare the new class to other source modules. It contains all the information they need to know about the class:

- Through its list of method declarations, the interface file lets other modules know what messages can be sent to objects belonging to the class. Every method that can be used outside the class definition is declared in the interface file; methods that are internal to the class implementation can be omitted, provided they're defined before they're used.
- The interface file also lets potential subclasses of the class know what instance variables they'll inherit, and how they'll be linked into the inheritance hierarchy.

Separating an object's interface from its implementation fits well with the design of object-oriented programs. An object is a self-contained entity that can be viewed from the outside almost as a "black box." Once you've determined how an object will interact with other elements in your program—that is, once you've declared its interface—you can freely alter its implementation without affecting any other part of the application.

Importing the Interface

The interface file must be included in any source module that mentions the class. It's usually included with the **#import** directive:

```
#import "Matrix.h"
```

This directive is identical to **#include**, except that it makes sure that the same file is never included more than once. It's therefore preferred, and is used in place of **#include** in source code throughout this manual.

Since the interface file itself mentions another class—its own superclass—it begins by importing the interface file for the superclass:

```
#import "ItsSuperclass.h"

@interface ClassName : ItsSuperclass
{
    variableDeclarations
}
methodDeclarations
@end
```

This convention means that every interface file includes, indirectly, the interface files for all inherited classes.

The Implementation

A class definition is structured very much like its declaration. It begins with an **@implementation** directive and ends with **@end**:

```
@implementation ClassName : ItsSuperclass
{
    variableDeclarations
}
methodDefinitions
@end
```

However, every implementation file imports its own interface. For example, **Matrix.m** imports **Matrix.h**. Because the implementation doesn't need to repeat any of the declarations it imports, it can safely omit:

- The declaration of a superclass
- The declarations of instance variables

This simplifies the implementation and makes it mainly devoted to method definitions:

```
#import "ClassName.h"

@implementation ClassName
methodDefinitions
@end
```

Methods for a class are defined, like C functions, within a pair of braces. Before the braces, they're declared in the same manner as in the interface file. For example:

```
+ new
{
}

- display
{
}

- moveTo: (NXCoord)x : (NXCoord)y
{
}
```

The definition of an instance method has all the instance variables of a potential receiving object within its scope. It can refer to them simply by name. Although the compiler creates the equivalent of C structures to store instance variables, the exact nature of the structure is hidden. You don't need the structure member operator (.) or the structure pointer operator (->) to refer to an object's data. For example, the following method definition refers to the receiver's **tag** instance variable:

```
- setTag: (int)anInt
{
    tag = anInt;
}
```

As the next example shows, instance variables and methods can share the same name:

```
- (float)backgroundGray
{
    return (backgroundGray);
}
```

Methods that take a variable number of arguments handle them just as a functions would:

```
#import <stdarg.h>

- getGroup:group, ...
{
    va_list ap;
    va_start(ap, group);
    ...
}
```

Adding to a Class

Methods can be added to a class by declaring them in an interface file under a category name and defining them in an implementation file under the same category. The category name indicates that the methods are additions to an existing class, not a new class.

The declaration of a category interface must import the interface file for the class that's being extended:

```
#import "ClassName.h"

@interface ClassName (Category)
methodDeclarations
@end
```

Note that a category can't declare any new instance variables for the class; it includes only method declarations.

The implementation, as usual, imports its own interface. Assuming that interface and implementation files are named after the category, it looks like this:

```
#import "Category.h"

@implementation ClassName (Category)
methodDefinitions
@end
```

The methods added in a category can be used to extend the functionality of a class or override methods the class inherits. However, they can't override methods defined elsewhere in the class; a class can't define the same method more than once. There's no limit to the number of categories that you can add to a class, but each category name must be different, and each must declare and define a different set of methods.

Categories can be used to extend classes defined by other implementors—for example, you can add methods to the classes defined in the NeXT software kits. The added methods will be inherited by subclasses and will be indistinguishable at run time from the original methods of the class.

Categories can also be used to distribute the implementation of a new class into separate source files—for example, you could group the methods of a large class into several categories and put each category in a different file. When used like this, categories can benefit the development process in a number of ways:

- They provide a simple way of grouping related methods. Similar methods defined in different classes can be kept together in the same source file.
- They simplify the management of a large class when more than one developer is contributing to the class definition.

- They let you achieve some of the benefits of incremental compilation for a very large class.
- They can help improve locality of reference for commonly used methods.
- They enable you to configure a class differently for different applications, without having to maintain different versions of the same source code.

Variables and Class Objects

Every instance of a class has its own copy of all the instance variables declared for the class; each object controls its own data.

The class object, on the other hand, has no variables. Only internal data structures are provided for the class; there are none that you can set or access directly. The class object also has no access to any instance variables; it can't initialize, read, or alter them.

Therefore, for all the instances of a class to share data, an external variable of some sort is required. Some classes declare static variables and provide class methods to manage them. (Declaring a variable **static** in the same file as the class definition limits its scope to just the class—and to just the part of the class that's implemented in the file. Unlike instance variables, static variables can't be inherited by subclasses.)

Static variables help give the class object more functionality than just that of a “factory” producing instances; it can approach being a complete and versatile object in its own right. A class object can be used to coordinate the instances it creates, allocate instances from lists of objects already created, or manage other processes essential to the application. In the limiting case, when you need only one object of a particular class, you can put all the object's state into static variables and use only class methods. This saves the step of creating an instance.

Note: It would also be possible to use external variables that weren't declared **static**, but the limited scope of static variables better serves the purpose of encapsulating data into separate objects.

How Messaging Works

In Objective-C, messages aren't bound until run time. The compiler converts a message expression,

[receiver message]

into a call on a messaging function, `objc_msgSend()`. This function takes the receiver and the name of the method mentioned in the message—that is, the method selector—as its two principal arguments:

`objc_msgSend(receiver, selector)`

Any arguments passed in the message are also handed to `objc_msgSend()`:

`objc_msgSend(receiver, selector, arg1, arg2, . . .)`

The messaging function does everything necessary for dynamic binding:

- It first finds the procedure (method implementation) that the selector refers to. Since the same method can be implemented differently by different classes, the precise procedure that it finds depends on the class of the receiver.
- It then calls the procedure, passing it the data structure (instance variables) of the receiving object, along with any arguments that were specified for the method.
- Finally, it passes on the return value of the procedure as its own return value.

Note: The compiler generates calls to the messaging function. You should never call it directly in the code you write.

The key to messaging lies in the structures that the compiler builds for each class and object. Every class structure includes these two essential elements:

- A pointer to the superclass.
- A class *dispatch table*. This table has entries that associate method selectors with the addresses of the methods they name. The selector for the `moveTo::` method is associated with the address of (the procedure that implements) `moveTo::`, the selector for the `display` method is associated with `display`'s address, and so on.

When a new object is created, its instance variables are stored in a memory location identified by the object name. Among the object's variables is a pointer to its class structure. This pointer, called `isa`, serves to identify the object's class.

These elements of class and object structure are illustrated in Figure 3-2.

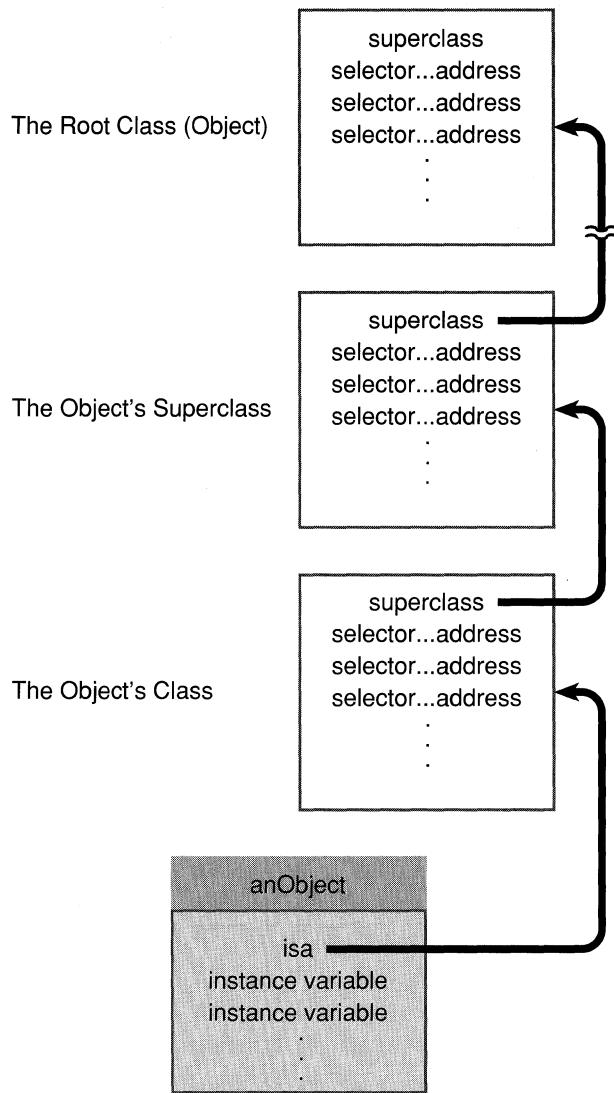


Figure 3-2. Messaging

When a message is sent to an object, the messaging function follows the object's `isa` pointer to the class structure, where it looks up the method selector in the dispatch table. If it can't find the selector there, `objc_msgSend()` follows the pointer to the superclass and tries to find the selector in its dispatch table. Successive failures cause `objc_msgSend()` to climb the class hierarchy until it reaches the `Object` class. Once it locates the selector, `objc_msgSend()` calls the method entered in the table and passes it the receiving object's data structure.

To speed the messaging process, the run-time system caches the selectors of methods currently in use. There's a separate cache for each class, and it can contain selectors for inherited methods as well as for methods defined in the class. Before searching the dispatch tables, the messaging routine checks the cache of the receiving object's class. If the method selector is in the cache, messaging is only slightly slower than a function call. Once a

program has been running long enough to “warm up” its caches, almost all the messages it sends will find a cached method. Caches grow dynamically to accommodate new messages as the program runs.

Selectors

For efficiency, full ASCII names are not used as method selectors in compiled code. Instead, the compiler and run-time system write each method name into a table, then pair the name with a unique identifier (an **unsigned int**) that will serve as its proxy. Compiled selectors are assigned to a special data type, SEL, to distinguish them from other integers.

A compiled selector contains fields of coded information that aid run-time messaging. You should therefore let the system assign SEL identifiers to methods; it won’t work to assign them arbitrarily yourself.

The **@selector()** directive lets Objective-C source code refer to the compiled selector, rather than to the full method name. Here the selector for **moveTo::** is assigned to the **action** variable:

```
SEL action;
action = @selector(moveTo::);
```

Values generally should be assigned to SEL variables at compile time with the **@selector()** directive. However, in some cases, a program may need to convert a character string to a selector at run time. This can be done with the **sel_getUid()** function:

```
action = sel_getUid("moveTo::");
```

This and other run-time functions are described in the *NeXTstep Reference* manuals.

Varying the Message at Run Time

The **perform:**, **perform:with:**, and **perform:with:with:** methods, defined in the Object class, take SEL identifiers as their initial arguments. All three methods map directly into the messaging function. For example,

```
[myMatrix perform:@selector(moveTo::) with:30.0 with:50.0]
```

is equivalent to:

```
[myMatrix moveTo:30.0 :50.0];
```

These methods make it possible to vary a message at run time, just as it's possible to vary the object that receives the message. Variable names can be used in both halves of a message expression:

```
id    target = [anObject getTheReceiver];
SEL   action = [anObject getTheSelector];
[target perform:action];
```

In this example, the receiver (**target**) is chosen at run time (by the fictitious **getTheReceiver** method), and the method the receiver is asked to perform (**action**) is also determined at run time (by the imaginary **getTheSelector** method).

perform: and its companion methods return an **id**. If the method that's performed returns a different type, it should be cast to the proper type:

```
float myGray;
myGray = (float) [target perform:@selector(backgroundGray)];
```

Identifying Return and Argument Types

Compiled selectors identify method names, not method implementations. Matrix's **backgroundGray** method, for example, will have the same selector as **backgroundGray** methods defined in other classes. This is essential to polymorphism; it lets methods like **perform:** and **perform:with:** send the same message to receivers belonging to different classes. (However, identically named class and instance methods have different compiled selectors.)

The messaging routine has access to method implementations only through selectors, so it treats all methods with the same selector alike. It discovers the return type of a method, and the data types of its arguments, from the selector. Therefore, except for messages sent to statically typed receivers, dynamic binding requires all implementations of identically named methods to have the same return type and the same argument types. (Statically typed receivers are an exception to this rule, since the compiler can learn about the method implementation from the class type.)

Hidden Arguments

When the messaging function finds the procedure that implements a method, it calls the procedure and passes it all the arguments in the message. It also passes the procedure two hidden arguments:

- The receiving object. This argument is how the receiver's instance variables come within the scope of the method implementation.
- The selector for the method.

These arguments are “hidden” because they aren’t declared in the source code that defines the method. They’re inserted into the implementation when the code is compiled.

Although these arguments aren’t explicitly declared, source code can still refer to them (just as it can refer to the receiving object’s instance variables). Methods refer to the receiving object as **self**, and to their own selectors as **_cmd**. In the example below, **_cmd** refers to the selector for the **strange** method and **self** to the object that receives a **strange** message.

```
- strange
{
    id target = [anObject getTheReceiver];
    SEL action = [anObject getTheMethod];

    if ( target == self || action == _cmd )
        return nil;
    return [target perform:action];
}
```

self is the more useful of the two arguments. It’s discussed in more detail in the next section.

Messages to self and super

Messaging syntax poses a problem whenever a method needs to refer to the object that performs it. Suppose, for example, that the **reposition** method needs to change the coordinates of whatever object it acts on. It can use the **moveTo::** method to make the change, but it can’t know the exact name of the receiving object. The object that receives the **moveTo::** message should be the very same object that the **reposition** message itself is sent to. But a **reposition** message might be sent to any number of objects with any number of different names:

```
- reposition
{
    [??? moveTo:30.0 :50.0];
    .
}
```

To solve this problem, Objective-C provides two terms that refer to the object that performs a method:

self
super

The **reposition** method should read either:

```
- reposition
{
    [self moveTo:30.0 :50.0];
    .
}
```

or:

```
- reposition
{
    [super moveTo:30.0 :50.0];
    .
}
```

Here **self** and **super** both refer to the object receiving a **reposition** message, whatever object that may happen to be. The two terms are quite different, however. **self** is one of the hidden arguments that the messaging routine passes to every method; it's a local variable that can be used freely within a method implementation, just as the names of instance variables can be. **super** is a term that substitutes for **self** only as the receiver in a message expression. As receivers, the two terms differ principally in how they affect messaging:

- **self** searches for the method implementation in the usual manner, starting in the dispatch table of the receiving object's class. In the example above, it would begin with the class of the object receiving the **reposition** message.
- **super** starts the search for the implementation in a very different place. It begins in the superclass of the class that defines the method where **super** appears. In the example above, it would begin with the superclass of the class where **reposition** is defined.

Wherever **super** receives a message, the compiler substitutes another messaging routine for **objc_msgSend()**. The substitute routine looks directly to the superclass of the defining class—that is, to the superclass of the class sending the message to **super**—rather than to the class of the object receiving the message.

An Example

The difference between **self** and **super** becomes clear in a hierarchy of three classes. Suppose, for example, that we create an object belonging to a class called **Low**. **Low**'s superclass is **Mid**; **Mid**'s superclass is **High**. All three classes define a method called **setRadius**. **Mid** also defines a method called **drawCircle**, which uses the **setRadius** method:

Classes	Methods
High	setRadius
Mid	setRadius, drawCircle
Low	setRadius

We now send a message to our Low object to apply the **drawCircle** method, and **drawCircle**, in turn, sends a **setRadius** message to the same Low object. If **drawCircle** calls this object **self**,

```
- drawCircle
{
    [self setRadius];
    .
    .
}
```

the messaging routine will find the version of **setRadius** defined in Low, **self**'s class. However, if **drawCircle** calls this object **super**,

```
- drawCircle
{
    [super setRadius];
    .
    .
}
```

the messaging routine will find the version of **setRadius** defined in High. It ignores the receiving object's class (Low) and skips to the superclass of Mid, the class where **drawCircle** is defined. Neither message finds Mid's version of **setRadius**.

As this example illustrates, **super** provides a way to bypass a method that overrides another method. Here it enabled **drawCircle** to avoid the Mid and Low versions of **setRadius** that redefined the original High version.

Not being able to reach Mid's version of **setRadius** may seem like a flaw, but, under the circumstances, you wouldn't ever want to perform it:

- The author of the Low class intentionally overrode Mid's version of **setRadius** so that instances of the Low class (and its subclasses) would apply the redefined version of the method instead.
- In sending the message to **super**, the author of Mid's **drawCircle** method intentionally skipped over Mid's version of **setRadius** (and over any versions that might be defined in Mid's subclasses) to perform the version defined in the High class.

Using **super**

Messages to **super** allow functionality to be distributed over more than one class. You can override an existing method to modify or add to it, and still incorporate the original method in the modification:

```
- (float)backgroundGray
{
    .
    .
    return([super backgroundGray]);
}
```

For some tasks, each class in the inheritance hierarchy can implement a method that does part of the job, and pass the message on to **super** for the rest.

It's also possible to concentrate core functionality in one method defined in a superclass, and have subclasses incorporate the method through messages to **super**. For example, every class method that creates a new instance must allocate storage for the new object and initialize its **isa** pointer to the class structure. This is typically left to the **new** method defined in the Object class. Methods defined in other classes are linked to Object's method, directly or indirectly, through messages to **super**. A typical instance-creating class method is shown below. Note that it declares a local variable to hold the new instance and statically types it to the name of the class:

```
+ new
{
    ClassName *newInstance;

    newInstance = [super new];
    [newInstance initObject];
    return(newInstance);
}
```

This hypothetical **new** method first performs a **new** method defined in another class higher up the inheritance hierarchy. It then initializes the new object by sending it an **initObject** message, and finally returns the object it created. The **new** method farther up the hierarchy might be Object's **new** method, or it might be a method with its own message to **super**. In either case, the chain of messages will reach up to the Object class.

Redefining self

super is simply a flag to the compiler telling it where to begin searching for the method to perform, but **self** is a variable name that can be assigned a new value. The new method shown above is a class method, so, initially, **self** and **super** both refer to the class object. In an instance method, they would both refer to the instance.

To illustrate the special role of **self**, let's suppose that the **initObject** method used by **new** in the example above did just two things—perform the **setRadius** method and initialize an instance variable, **numCount**.

```
- initObject
{
    [self setRadius];
    numCount = 1;
}
```

Suppose also that we wanted to move these operations into the **new** method. Two changes would have to be made:

- The **setRadius** method would have to be sent to **newInstance** rather than **self**. In the **new** method, **self** refers to the class object, and **setRadius** is an instance method.
- The direct reference to the **numCount** instance variable would have to become indirect. When an instance variable is mentioned in a method, it's assumed to belong to **self**. For **new**, **self** is the class object, and classes don't have instance variables.

These two changes are shown in the revised example below:

```
+ new
{
    ClassName *newInstance;

    newInstance = [super new];
    [newInstance setRadius];
    newInstance->numCount = 1;
    return(newInstance);
}
```

Note: The indirect reference to **numCount** as a member of the **newInstance** structure is possible here only because **newInstance** is statically typed to be a **ClassName** object and is declared within the implementation of the **ClassName** class. See “Options” later for more on static typing.

The two coding changes illustrated above are necessary only because, in a class method like **new**, **self** refers to the class object. But **self** is a variable; it can be redefined. Class methods often redefine it to be the new instance. This would simplify the **new** method shown above, and let it absorb **initObject**'s code unchanged:

```
+ new
{
    self = [super new];
    [self setRadius];
    numCount = 1;
    return(self);
}
```

Redefining **self** doesn't affect the meaning of **super**; it still refers to the class object. If the **setRadius** message in the example above were sent to **super** instead of **self**, the run-time system would search for a class method named **setRadius**, rather than an instance method.

The Object Class

Because the **Object** class is at the root of all inheritance hierarchies, the methods it defines are inherited by every other class. Its methods therefore define behaviors that are inherent to every instance and class object. A few of these methods—such as, **new**, **class** and **perform**:—have already been mentioned in this chapter. Others are discussed here. For complete information on the **Object** class, see *NeXTstep Reference, Volume 1*.

Memory Management

Objects are created at run time in dynamically allocated memory. When an object has outlived its usefulness, this memory should be freed. Object's **free** method releases the memory occupied by the receiver:

```
[self free];
```

Future messages to the receiver will result in an error.

Note: Class objects are created at compile time and can't be freed.

Assigning an object to another variable, as in this example

```
id target;
target = myMatrix;
```

merely stores its **id** under a different name; it doesn't duplicate the object. To duplicate an object, memory must be dynamically allocated for a new instance. The **copy** method allocates this memory and returns the copy:

```
id myClone;
myClone = [myMatrix copy];
```

Class Initialization

The run-time system sends an **initialize** message to every class object before the class receives any other messages. This gives the class a chance to set up its run-time environment before it's used. If no initialization is required, you don't need to write an **initialize** method to respond to the message; the Object class defines an empty version that your class can inherit and apply.

If a class makes use of static or global variables, the **initialize** method is a good place to set their initial values. Some classes in the NeXT software kits define **initialize** methods to read in values for program parameters from the user's defaults database. (For more information on these parameters and the database, see Chapter 10, "Support Objects and Functions.")

Avoiding Messaging Errors

If an object receives a message to perform a method that isn't in its repertoire, an error results. It's the same sort of error as calling a nonexistent function. But because messaging occurs at run time, the error often won't be evident until the program executes.

It's relatively easy to avoid this error when the message selector is constant and the class of the receiving object is known. As you're programming, you can check to be sure that the receiver is able to respond. The compiler will check for you if the receiver is statically typed.

However, if the message selector or the class of the receiver varies, it may be necessary to postpone this check until run time. The **respondsTo:** method, defined in the Object class, determines whether a potential receiver can respond to a potential message. It takes the method selector as an argument, and returns whether the receiver has access to a method matching the selector:

```
if ( [target respondsToSelector:@selector(moveTo::)] )
    [target moveTo:0.0 :0.0];
else
    fprintf(stderr, "target can't be moved");
```

Archiving Support

The Object class has support for copying object data structures from one location to another. An object can be stored in a file and later reactivated, for example, or sent to another application, which can then activate and use it.

The **write:** method writes an object (its instance variables) to a data stream, and **read:** reads in an object from a stream, reinitializing its instance variables. The stream is generally connected to an archive file where the object is stored, but it might be connected to a port, to memory, or to some other repository.

Any class that declares instance variables must define its own **read:** and **write:** methods to read and write the instance variables declared in the class. So that all of an object's instance variables can be read and written, each new version of either method should incorporate, through a message to **super**, the version it overrides:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    . . .
}
```

Writing an object to a stream is initiated by a call to the **NXWriteObject()** function, which sends the object a **write:** message. Reading an object from a stream is initiated by a call to **NXReadObject()**, which allocates memory for a new object of the correct class and sends it a **read:** message to initialize its instance variables from the stream. Although you may implement versions of **read:** and **write:**, **read:** and **write:** messages should be generated only indirectly through **NXReadObject()** and **NXWriteObject()**.

See Chapter 10 for more on archiving, streams, and these functions.

Immediately after an object has been unarchived with the **read:** method, **NXReadObject()** sends it an **awake** message. The inherited version of **awake** defined in the Object class does nothing but return **self**. But a class can define an **awake** method of its own to reinitialize its instances and make sure they're in a usable state before they receive any other messages. For example, after a Window object is unarchived, its **awake** method gets a window for it to manage from the Window Server.

After the **awake** message, **NXReadObject()** sends each unarchived object a **finishUnarchiving** message. This message gives Objective-C programs a chance to free the unarchived object and substitute another object for it. **finishUnarchiving** should return **nil** if there is no substitution, and the replacement object if there is. The default version defined in the Object class returns **nil**.

Inheritance Relationships

Every object can identify its position in the inheritance hierarchy. The **class** method returns the **id** of the receiver's class and **name** returns the class name:

```
id           myClassObject = [myMatrix class];
const char *myClassName = [myMatrix name];
```

The **superClass** method returns the **id** of the receiver's superclass:

```
id myParent = [myMatrix superClass];
```

Object's **isMemberOf:** and **isMemberOfGivenName:** methods test whether the receiver is an instance of a particular class. In these examples, they test whether **myMatrix** is an instance of **View**. It isn't, so both messages would return NO:

```
BOOL idTest = [myMatrix isMemberOf:[View class]];
BOOL nameTest = [myMatrix isMemberOfGivenName:@"View"];
```

Two similar methods, **isKindOf:** and **isKindOfGivenName:** test whether the receiver inherits from a particular class. Since a Matrix is a kind of View (the Matrix class inherits from View), the two messages below would return YES:

```
BOOL idTest = [myMatrix isKindOf:[View class]];
BOOL nameTest = [myMatrix isKindOfGivenName:@"View"];
```

The “is kind of” and “is member of” relationships are both basic to inheritance in object-oriented programming. For example, an object can be statically typed to any class that it's a “kind of,” but responds to messages with all the methods defined for the class it's a “member of.”

Posing

You can have a subclass take the place of its own superclass at run time. The subclass inherits from its superclass, so none of the functionality of the superclass is lost in the substitution. But the subclass can add functionality to the superclass by defining new methods, or modify the superclass by overriding inherited methods. (However, it can't define any new instance variables of its own.)

In the example below, the fictional NewMatrix class takes the place of the Matrix class.

```
[NewMatrix poseAs:[Matrix class]];
```

A **poseAs:** message should be sent before the superclass receives any messages. After the **poseAs:** message is sent, all messages sent to the superclass will actually be received by the posing subclass. The subclass can also receive messages under its own name.

Posing is a way of adding methods to an existing class definition. A simpler and more direct way to add methods is to implement them in a category of the class, not in a subclass. A class can have any number of categories, but only one posing subclass. Categories also don't require you to send a message to have them take effect. (See "Adding to a Class" earlier in this chapter for information on categories.)

There's only one thing that posing can do that a category can't: A posing subclass can override methods defined in the superclass it replaces, but a category can't override methods already defined for the class it extends.

Options

Objects are dynamic entities. As many decisions about them as possible are pushed from compile time to run time:

- The memory for objects is *dynamically allocated* at run time by class methods that create new instances.
- Objects are *dynamically typed*. In source code (at compile time), all objects are of type **id**. The exact class of an **id** variable (and therefore its particular data structure) isn't determined until the program is running.
- Objects and methods are *dynamically bound*, as described under "How Messaging Works" above. A run-time procedure locates the method that "belongs to" the object and passes it the object's instance variables.

These features give object-oriented programs a great deal of flexibility and power, but there is a price to pay. Messages are somewhat slower than function calls, for example, and the compiler can't check the exact types (classes) of **id** variables.

Since, on occasion, particular features of object-oriented programming may be less important than speed and directness, Objective-C lets you turn some of its object-oriented features off in order to shift operations from run time to compile time.

Static Typing

If a class name is used in place of **id** in an object declaration,

```
Matrix *thisObject;
thisObject = [Matrix new];
```

it restricts the declared variable to instances of the class and its subclasses. In the example above, **thisObject** can only be a **Matrix** of some kind.

Although **thisObject** is statically typed, it's still dynamically allocated by the same class method that creates instances of type **id**. Because an **id** is really a pointer to an object, **thisObject** is declared as a pointer to a **Matrix**.

Statically typed objects have the same internal data structures as objects declared to be **ids**. The type doesn't affect the object; it affects only the amount of information given to the compiler to refer to the object. With the additional information provided by static typing, the compiler can deliver better type-checking services in two situations:

- When a message is sent to a statically typed receiver, the compiler can check to be sure that the receiver can respond. A warning is issued if the receiver doesn't have access to the method named in the message.
- When a statically typed object is assigned to a statically typed variable, the compiler can check to be sure that the types are compatible. A warning is issued if they're not.

An assignment can be made without warning provided the class of the object being assigned is identical to, or inherits from, the class of the variable receiving the assignment. This is illustrated in the example below.

```
View *aView;
Matrix *aMatrix;

aMatrix = [Matrix new];
aView = aMatrix;
/* aMatrix = aView;           this doesn't work */
```

Here **aMatrix** can be assigned to **aView** because a **Matrix** is a kind of **View**—the **Matrix** class inherits from **View**. However, if the roles of the two variables are reversed and **aView** is assigned to **aMatrix**, the compiler will generate a warning; not every **View** is a **Matrix**.

There's no check when the expression on either side of the assignment operator is an **id**. A statically typed object can be freely assigned to an **id**, or an **id** to a statically typed object:

```
id      firstObject, secondObject;
Matrix *aMatrix;

firstObject = [Matrix new];
aMatrix = firstObject;
secondObject = aMatrix;
```

Because methods like **new** return **ids**, the compiler doesn't check to be sure that a compatible object is returned to a statically typed variable. The following code is error-prone, but is allowed nonetheless:

```
Matrix *aMatrix;
aMatrix = [Window new];
```

Note: This is consistent with the implementation of **void *** (pointer to void) in ANSI C. Just as **void *** is a generic pointer that eliminates the need for coercion in assignments between pointers, **id** is a generic pointer to objects that eliminates the need for coercion to a particular class in assignments between objects.

Better type checking is just one of the advantages of static typing. When an object is statically typed, these other options also become possible:

- The object's instance variables can be declared public and accessed directly.
- The object can be freed from the restriction that identically named methods must have identical return and argument types.

These two topics are discussed in the sections that follow.

Public Instance Variables

The instance variables of a statically typed object can be made public, instead of being treated as private to the object. Rather than send a message (like **backgroundGray**) to the object asking it to reveal a value, you can access public instance variables directly, as components in a structure:

```
thisGray = thisObject->backgroundGray;
```

In general, this directness is possible only for instance variables that have been declared public in the interface file. All instance variables that are declared after an **@public** directive are considered public in statically typed objects. The example below shows how the Matrix class could have made its **backgroundGray** instance variable public:

```
@interface Matrix : Control
{
    private variables
@public
    float backgroundGray;
    other public variables
}
```

Note: None of the instance variables declared in the NeXT software kits are public.

There's just one context in which the **@public** directive isn't needed to make an instance variable public. A statically typed instance of a class has public instance variables within the implementation of its own class. For example, in the implementation of the Foo class, a method could create a Foo instance and directly access its instance variables:

```
- getCounterpart
{
    Foo *aFoo = [self copy];
    aFoo->variable = 0;
    . . .
}
```

This feature was used in the presentation of the **new** method under “Messages to **self** and **super**” above.

Return and Argument Types

In general, methods that share the same selector (the same name) must also share the same return and argument types. This constraint is imposed by dynamic binding because the class of the message receiver, and therefore class-specific details about the method it's asked to perform, can't be known at compile time.

However, when a message is sent to a statically typed object, the class of the receiver is known by the compiler. Therefore, the message is freed from the restrictions on its return and argument types.

Getting a Method Address

To dynamically bind a receiving object to a method implementation, the messaging routine must find the implementation that's appropriate for the object. This was discussed under “How Messaging Works” above.

On occasion, as when a particular method will be performed many times in succession, you may want to avoid repeating this step each time the method is performed. With a method defined in the Object class, **methodFor:**, you can ask for a pointer to the procedure that implements a method, then use the pointer to call the procedure. The example below shows how the procedure that implements the **moveTo::** method might be called:

```
id    (*matrixMover) ();
int   i;

matrixMover = [target methodFor:@selector(moveTo:)];
for (i = 1000; i > 0; i--)
    (* matrixMover)(target, @selector(moveTo:),
                    (NXCoord)x++, (NXCoord)y++);
```

The procedure call is indirect, here through the **matrixMover** pointer returned by **methodFor:**. The first two arguments passed to the procedure are the receiving object (**self**) and the method selector (**_cmd**).

As illustrated above, **methodFor:** always returns a pointer to a function that returns an **id**. If the function actually returns another value (as the procedure that implements **backgroundGray** would), you must coerce the value returned by **methodFor:** to the correct type:

```
float (*grayGetter)();

grayGetter = (float (*)()) [target
                           methodFor:@selector(backgroundGray)];
```

Using **methodFor:** to circumvent dynamic binding saves most of the time required by messaging. However, the savings will be significant only where a particular message would be repeated many times, as in the **for** loop shown above.

Getting an Object Data Structure

A fundamental tenet of object-oriented programming is that the data structure of an object is private to the object. Information stored there can be accessed only through messages sent to the object. However, there's a way to strip an object data structure of its “objectness” and treat it like any other C structure. This makes all the object's instance variables publicly available.

When given a class name as an argument, the `@defs()` directive produces the declaration list for an instance of the class. This list is useful only in declaring structures, so `@defs()` can appear only in the body of a structure declaration. This code, for example, declares a structure that's identical to the template for an instance of the `Matrix` class:

```
struct matrixDef {
    @defs(Matrix)
} *public;
```

Here `public` is declared as a pointer to a structure that's essentially indistinguishable from a `Matrix` instance. With a little help from a type cast, a `Matrix` `id` can be assigned to the pointer. The object's instance variables can then be accessed publicly through the pointer:

```
id aMatrix;
aMatrix = [Matrix new];

public = (struct matrixDef *)aMatrix;
public->backgroundGray = 0.0;
```

Type Encoding

To assist tools in the run-time environment, the compiler encodes the return and argument types for each method in a short string and associates the string with the method selector. The coding scheme it uses can also be of use in other contexts and so is made publicly available with the `@encode()` directive. When given a type specification, `@encode()` returns the string encoding that type. The type can be a basic type such as an `int`, a pointer, a tagged `struct` or `union`, or a class name—anything, in fact, that can be used as an argument to the C `sizeof()` operator.

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(Matrix);
```

The table on the next page lists the type codes. Note that many of them overlap with the codes used in writing to a typed stream. However, there are codes listed here that you can't use when writing to a typed stream and there are codes that you may want to use when writing to a typed stream that aren't generated by `@encode()`. (See Chapter 10 for information on typed streams.)

Code	Meaning
c	A char
i	An int
s	A short
l	A long
C	An unsigned char
I	An unsigned int
S	An unsigned short
L	An unsigned long
f	A float
d	A double
v	A void
*	A character string (char *)
@	An id
#	A pointer to a class (Class)
:	A method selector (SEL)
[...]	An array
{...}	A structure
(...)	A union
bnum	A bitfield of <i>num</i> bits
$^{\text{type}}$	A pointer to <i>type</i>
?	An unknown type

The type specification for an array is enclosed within square brackets; the number of elements in the array is specified immediately after the open bracket, before the array type. For example, an array of 12 pointers to **floats** would be encoded as:

```
[12^f]
```

Structures are specified within braces, and unions within parentheses. The type of each component within the structure or union is listed in sequence, even for structures and unions that are **typedef**'ed. However, for pointers to structures and unions, only the structure or union tag is listed. For example, an NXRect structure (discussed in the next chapter) would be encoded as:

```
{ {ff} {ff} }
```

But a pointer to an NXRect structure would be encoded using the structure tag:

```
^{_NXRect}
```

This avoids the endless recursion that would otherwise result when a structure component points to a structure of the same type.

A question mark indicates an unknown type. It's generated mainly by untagged structures and unions and by pointers to functions.

Language Synopsis

Objective-C adds a small number of constructs to the C language and defines a handful of conventions for effectively interacting with the run-time system. Most of these constructs and conventions were described in the preceding sections of this chapter. This section summarizes all the additions to the language. It mentions only one or two topics that weren't discussed earlier; these topics play a less central role in the construction of object-oriented programs.

For a more formal presentation of Objective-C syntax, see the grammar in the *NeXT Technical Summaries* manual.

Messages

Message expressions are enclosed in square brackets:

[*receiver message*]

The *receiver* can be:

- A variable or expression that evaluates to an object
- A class name (indicating the class object)
- **super** (indicating an alternative search for the method implementation)

The *message* is the name of a method plus any arguments passed to it.

Defined Types

The principal types used in Objective-C are defined in **objc/objc.h**. They are:

id	An object (a pointer to its data structure)
Class	A class (a pointer to its data structure)
SEL	An unsigned int that identifies a method
STR	A character string
IMP	A pointer to a method implementation that returns an id
BOOL	A boolean value, either YES or NO

(The **Class** type is specific to NeXT's implementation of the language.)

In addition, class names can be used to statically type instances of the class.

Preprocessor Directives

The preprocessor understands two new notations:

`#import` Imports a header file. This directive is identical to `#include`, except that it won't include the same file more than once.

`//` Begins a comment that continues to the end of the line.

Compiler Directives

Directives to the compiler begin with “@”:

<code>@selector(method)</code>	Returns the compiled selector that identifies <i>method</i> .
<code>@interface</code>	Begins the declaration of a class or category interface.
<code>@implementation</code>	Begins the definition of a class or category.
<code>@end</code>	Ends the declaration or the definition of a class or category.
<code>@public</code>	Precedes the declaration of public instance variables.
<code>@encode(spec)</code>	Yields a character string that encodes the type structure of <i>spec</i> .
<code>@defs(classname)</code>	Yields the internal data structure of <i>classname</i> instances.

Method Declarations

The following conventions are used in method declarations:

- A “+” precedes declarations of class methods.
- A “-” precedes declarations of instance methods.
- Arguments are declared after colons (:). Typically, a label describing the argument precedes the colon. Both labels and colons are considered part of the method name.
- Argument and return types are declared using the C syntax for type casting.
- The default return and argument type for methods is `id`, not `int` as it is for functions. By extension, `id` is also the default return type for message expressions.

Method Implementations

Each method implementation is passed two hidden arguments:

- The receiving object (**self**)
- The selector for the method (**_cmd**)

Within the implementation, both **self** and **super** refer to the receiving object. **super** replaces **self** as the receiver of a message to indicate that only methods inherited by the implementation should be performed in response to the message.

Other Keywords

The **objc/objc.h** header file also defines these useful terms:

nil	A null object pointer, (id)0
Nil	A null class pointer, (Class)0

Chapter 4

Drawing

4-4 Design Philosophy

4-6 The Screen

- 4-6 Pixels, Halftones, and Rectangular Coordinates
- 4-7 The NeXT Computer Screen
- 4-9 Screen Coordinates
- 4-10 Window Coordinates
 - 4-11 Modifying the Coordinate System
 - 4-11 View Coordinates
 - 4-12 Borders and Content Areas
- 4-13 Printing Coordinates
- 4-14 Rectangles
 - 4-15 Rectangle Geometry
 - 4-17 When a Rectangle Isn't Rectangular

4-19 The Window System

- 4-20 Window Numbers
- 4-20 Contexts and Graphics States
 - 4-21 The Current Window
 - 4-22 Changing Graphics States
- 4-23 Window Buffering
- 4-25 Choosing a Buffering Type
- 4-26 The Screen List
 - 4-26 Window Tiers
 - 4-27 Off-Screen Windows
- 4-28 The Background Color

4-29 Compositing and Transparency

- 4-30 Transparent Paint
- 4-33 Data Representation
 - 4-33 Bitmaps
 - 4-34 Premultiplication
- 4-36 Compositing
 - 4-36 Compositing Operators
 - 4-36 `composite`
 - 4-38 `compositerect`
 - 4-39 Types of Compositing Operations
 - 4-41 Copy
 - 4-41 Clear
 - 4-42 PlusD and PlusL
 - 4-42 Transparency Operations

- 4-43 Dissolving
- 4-44 Highlighting

4-45 Instance Drawing

- 4-47 Sending PostScript Code to the Window Server**
- 4-48 Using **pswrap**
- 4-49 Using Single-Operator Functions
- 4-50 Connection Buffering

4-51 Imaging Conventions

- 4-52 The General Rule
- 4-56 Outlines with No Area
 - 4-56 Points
 - 4-57 Zero-Width Lines
- 4-59 Half-Open Shapes
- 4-60 Clipping

Chapter 4

Drawing

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/ReleaseNotes/WindowServer.rtf](#)
[/NextLibrary/Documentation/NextDev/ReleaseNotes/AppKit.rtf](#)

All of your program's visual output, whether sent to the laser printer or displayed on the screen, is generated through the PostScript language, a high-level, interpreted language with special facilities for drawing and handling graphic images, including text. The PostScript language has typically been used to produce high-quality images on the printed page. On the NeXT computer, it's also used for imaging on the screen. NeXT and Adobe Systems Incorporated have jointly developed the Display PostScript system, which refines the language and adapts it to the interactive requirements of the screen.

Because the NeXT computer creates both screen and printed images through the same language, the screen can display a document just as it will be printed. Except for differences in resolution, what you see is what you get.

Applications draw within windows—screen rectangles that can be moved, resized, and layered on top of each other—rather than directly on the screen. The NeXT window system is implemented as an extension to the Display PostScript system. At the most fundamental level, applications create and manage windows through PostScript operators. However, the Application Kit provides an object-oriented interface to the window system that most applications should use; the Kit handles basic window-management tasks for you.

NeXT has made other extensions to the Display PostScript system so that applications can better use the capabilities of the window system and MegaPixel Display. The principal additions permit compositing, drawing with partially transparent paint, and instance drawing:

- Compositing builds a new image by overlaying images that were previously drawn. It's analogous to a photographer printing a picture from two negatives, one placed on top of the other. However, on the NeXT computer there are even more ways that each of the images you start with can contribute to the composite.

All image-transferring operations, including simply copying an image from one location to another, are accomplished through compositing. Compositing is the only way to copy on the NeXT computer.

- Compositing can achieve interesting effects when the initial images are drawn with partially transparent paint. In a typical compositing operation, paint that's partially transparent won't completely cover the image it's placed on top of; some of the other image will show through. The more transparent the paint is, the more of the other image you'll see.

You can set the *coverage* of the paint you draw with—how transparent or opaque it is—just as you set its color or gray level.

- Instance drawing puts temporary images on the screen, and removes them again, at little processing cost. When instance drawing is removed, the original images are automatically restored. Instance drawing is useful for temporarily highlighting an image, for showing an object being dragged from one location to another, and for other kinds of dynamic drawing.

All operators added to the Display PostScript system by NeXT are described in detail in the *NeXTstep Reference, Volume 2*. The Display PostScript system is documented in four publications by Adobe Systems Incorporated—*PostScript Language Extensions for the Display PostScript System*, *Client Library Reference Manual*, *pswrap Reference Manual*, and *Perspective for Software Developers*. The PostScript language itself is described in the *PostScript Language Reference Manual* by Adobe Systems Incorporated, published by Addison-Wesley. A beginning guide to the language and a series of example graphics programs are provided in the *PostScript Language Tutorial and Cookbook*, by the same author and publisher.

This chapter discusses drawing on the NeXT computer using the Display PostScript system and its extensions. In addition to transparency, compositing, and instance drawing, it describes how drawing works in the NeXT window system and on the MegaPixel Display.

Later chapters—Chapters 6, “Program Structure,” 7, “Program Dynamics,” and 9, “User-Interface Objects”—discuss drawing in the broader context of the Application Kit. The Kit defines objects that draw the graphic elements of the NeXT user interface, including scrollers, switches, sliders, menus, and user-editable text; you don’t need to write this PostScript code yourself. You can therefore focus your attention on drawing that’s unique to your application. The drawing code you write will be integrated into the object-oriented program structure provided by the Kit, and will use the Display PostScript system and the extensions to it discussed in this chapter.

Design Philosophy

When one imaging model is used for the printer and another for the screen, application programs must pursue two parallel lines of development. In addition to the extra work, it’s nearly impossible to make screen and printed images match.

The NeXT computer avoids these problems by using a single imaging model, the PostScript language, for all drawing. The PostScript language is well-suited to this role because:

- It's device independent.
- It's programmable.
- It provides a complete two-dimensional imaging model.
- It's a widely used standard on printers.

Extensions to the PostScript language are the foundation for the NeXT window system. Drawing instructions and window management operations are both sent to the same interpreter. Because of this integration, you'll have fewer issues to worry about as you program your application.

Drawing on the screen has dynamic aspects that are missing when drawing for the printer:

- It's interactive. Applications must follow the user's instructions and respond graphically to the user's actions.
- It changes over time. Applications can repeatedly reuse the same area of a window by erasing and replacing what's displayed there.
- It uses a number of different windows simultaneously. Instead of drawing one page at a time, applications present the user with a multi-windowed interface.

These aspects present applications with both challenges and opportunities. One challenge is speed. To meet it, the Display PostScript system has been refined to respond quickly, without losing any of the original generality and power of the PostScript language. In a few cases, operators have been added so that common operations can be executed more efficiently.

Another challenge is programming simplicity. This challenge is met mainly by relieving applications of bothersome chores:

- Much of your program's drawing can be done through Application Kit objects that have the ability to draw themselves. Common drawing operations such as scrolling, resizing, clipping, and erasing are also handled through Application Kit objects. In addition, the Kit has facilities that make it relatively easy to use bitmaps and icons.
- The Window Server handles the dynamic behavior of windows. It moves and resizes them in response to user actions, without your program's intervention.
- The Window Server provides windows with backup buffers so that it can automatically save images when a window is hidden, and automatically restore them again when the window becomes visible once more.

Buffering also makes it possible to draw into windows that never appear on-screen. The images that are cached in off-screen windows can then be copied to windows on-screen, using the same operators that transfer images between on-screen windows.

NeXT has adopted an advanced model for moving and combining images. Compositing and transparency make it possible for programs to explore new visual effects, such as slowly dissolving one image into another and building a final image out of several layers. With window buffering, they make animation fairly simple. NeXT compositing is fully compatible with color graphics, so you won't have to redesign your program for a color screen. Compositing is implemented as an extension to the PostScript language, but it's also possible to composite in Objective-C code using the `Bitmap` object of the Application Kit.

To help applications provide immediate feedback to users, NeXT has also augmented the PostScript language with instance drawing. Images that are drawn in this mode are temporary; when they're removed, the original image reappears.

The Screen

The first part of this section briefly covers some background terminology. The parts that follow introduce drawing on the NeXT MegaPixel Display.

Pixels, Halftones, and Rectangular Coordinates

Images on the screen are formed from tiny picture elements, or *pixels*, arranged in a rectangular grid of columns and rows that fill the entire surface of the screen. Because of their positions in the grid, pixels can be thought of as little square dots. Each pixel has a separate representation in screen memory and can be assigned an independent color value. Varying the color of the individual pixels makes it possible to render an almost unlimited variety of images on the screen.

Typically, a pixel on a monochrome computer screen is capable of showing just two colors: either black or white. A black line is rendered by turning all the pixels along its path black; a white circle has all the pixels within its radius turned white. Intermediate shades of gray are rendered by *halftoning*, turning some pixels black and others white. A mixed pattern of adjacent black and white pixels appears as a solid, uniformly colored gray area. Figure 4-1 is a close-up diagram of a halftone gray area. The lines represent the pixel grid; the spaces between the lines represent individual pixels.

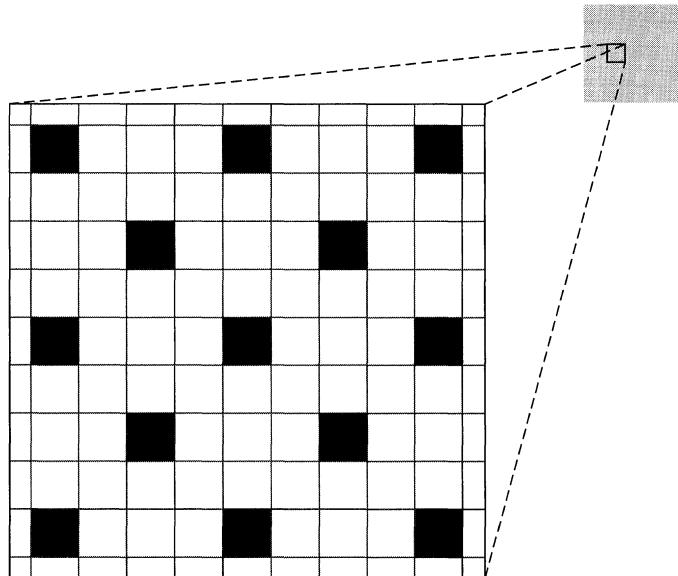


Figure 4-1. Halftone with Black and White Pixels

Locations on the screen are described using a standard rectangular (Cartesian) coordinate system. Points are identified by their position relative to a horizontal x-axis and a vertical y-axis. The origin, where the two axes meet, is (0, 0). A point 500 pixels to the right of the origin as measured along the x-axis and 200 pixels above the origin as measured along the y-axis is (500, 200). Negative coordinates locate points below the x-axis and to the left of the y-axis.

The *resolution* of an image is a function of pixel density. The more pixels there are in a given area (that is, the smaller they are), the more detailed and precise the image can be. Computer screens have a lower resolution (fewer pixels per inch) than do laser printers.

The NeXT Computer Screen

The visible area on the MegaPixel Display screen holds nearly a million pixels, 1120 along each horizontal row and 832 in each vertical column. There are about 92 pixels per inch in each direction (91.80 per inch along the x-axis and 92.44 along the y-axis when the display is exactly 12.2 inches by 9 inches, but these dimensions may vary by ± 0.100 inch).

Pixels on the NeXT monochrome screen aren't limited to just black and white; each one can also display two shades of gray, for a total of four discrete colors:

- white
- light gray
- dark gray
- black

This reduces the need for halftoning while increasing the number of halftone shades. Where halftones are used, there's less variation in the color of adjacent pixels, so the screen can show purer, less granular shades of gray. Transitions from one shade of gray to another in continuous-tone images are smoother. Contrasts between a shade of gray and either black or white are sharper, since the gray can be rendered with fewer, if any, black or white pixels.

Figure 4-2 shows the same shade of halftone gray as Figure 4-1, only rendered with light gray rather than black pixels. Note that Figure 4-2 has three times as many nonwhite pixels, all closer to the desired shade of gray than the black pixels in Figure 4-1.

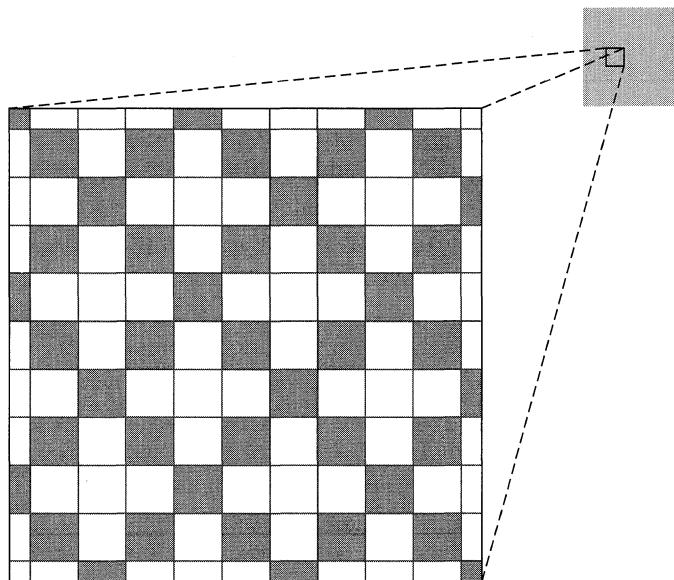


Figure 4-2. Halftone with Gray and White Pixels

Each pixel requires two bits of memory to distinguish the four colors. When needed, two additional bits are set aside to store the coverage of the pixel—how transparent or opaque it is. (See “Compositing and Transparency,” later in this chapter.) Halftoning produces intermediate “shades” of coverage, just as it produces intermediate shades of gray.

To avoid halftoning on the MegaPixel Display, you can set the gray level to a value that results in pure black, pure white, or a pure gray. These four C constants are defined to the required values:

```
NX_WHITE  
NX_LTGRAY  
NX_DKGRAY  
NX_BLACK
```

In the PostScript language, values specified to three decimal places work:

white	1.0
light gray	0.667
dark gray	0.333
black	0.0

Screen Coordinates

The entire screen can be thought of as the first (upper right) quadrant of a two-dimensional coordinate grid, with the origin in the lower left corner and the positive x-axis extending horizontally to the right and the positive y-axis extending vertically upward. A unit along either axis is equal to the distance across one pixel, approximately 1/92 inch.

Figure 4-3 illustrates this *screen coordinate system*.

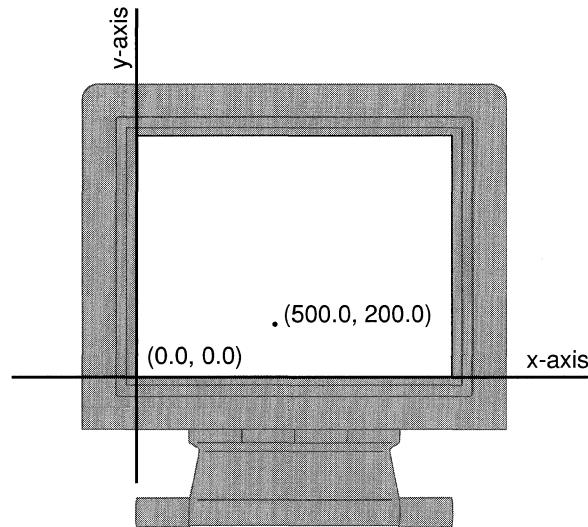


Figure 4-3. The Screen Coordinate System

Coordinates are specified as floating-point numbers. This permits you to move anywhere on the screen, not just from pixel to pixel. Coordinates that are integers—those that have no fractional part—lie between pixels in the screen coordinate system; fractional coordinates locate points somewhere within a pixel. Figure 4-4 indicates where four sample points are located in relation to pixel boundaries.

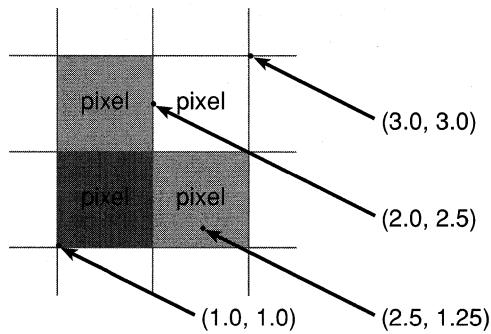


Figure 4-4. Points and Pixels

“Imaging Conventions” below explains how fractional coordinates and lines that cut through pixels are handled on-screen.

The screen coordinate system has just one function: to position windows on the screen. When your application creates a new window, it must specify the window’s initial size and location in screen coordinates. For drawing in a window, it will use a different coordinate system, one that’s specific to the window (see “Window Coordinates” below).

No matter where a window is located, its edges are always aligned on pixel boundaries. If you try to position a window using fractional coordinates, they will be adjusted to whole integers. Fractional coordinates can be freely used when drawing within a window, however.

As Figure 4-3 above illustrates, the screen coordinate system extends beyond the screen in all directions. However, its range is not unlimited. No screen coordinate (no point within a window) can have a value greater than 16000 or less than -16000:

$$(-16000 \leq x \leq +16000, -16000 \leq y \leq +16000)$$

All windows must fit within the space bounded by ± 16000 .

Window Coordinates

The reference coordinate system for a window is known as the *base coordinate system*. It differs from the screen coordinate system in only two ways:

- It applies only to a particular window; each window has its own base coordinate system.
- Its origin is at the lower left corner of the window, rather than the lower left corner of the screen. If the window moves, the origin and the entire coordinate system move with it. An image retains its position within a window’s coordinate system no matter where the window is located.

In PostScript language terms, the base coordinate system is the equivalent to a “default user space” for the window. It provides a starting place for programs that draw within a window, just as the default user space provides a starting place for programs that describe pages for a printer.

Note: You can also establish a reference coordinate system with the same coordinate units as the default user space, 1/72 inch. However, this documentation assumes the base coordinate system for all windows. The base coordinate system is more convenient for drawing on the screen, since coordinate units fall on pixel boundaries. It’s also the initial coordinate system established for all windows created through the Application Kit. (For more information, see the discussion of the **windowdeviceround** and **windowdevice** operators in the *NeXTstep Reference* manual.)

Modifying the Coordinate System

You’re not limited to the base coordinate system when you draw within a window. PostScript operators can radically transform the window’s coordinates. The origin can be moved (by the **translate** operator), the x- and y-axes can be turned to point in any direction (by the **rotate** operator), and the size of a unit along either the x- or y-axis can be expanded or shrunk (by the **scale** operator). The base coordinate system is simply the reference system for any subsequent transformations by PostScript operators.

PostScript transformations apply only within a window. They don’t alter the screen coordinate system, and therefore can’t affect the size or location of the window itself.

When you draw, coordinates are expressed in the application’s *current coordinate system*, the system reflecting the last coordinate transformations to have taken place within the current window. The PostScript interpreter keeps track of this system through the current transformation matrix (CTM) of the current graphics state.

View Coordinates

Drawing is usually limited to areas that are smaller than the whole window. The Application Kit permits you to set up rectangular regions within a window and to move and resize them, much as windows themselves are moved and resized on the screen. The Kit provides each region—or “View,” after the Objective-C class that defines them—with a coordinate system, which it makes the application’s current coordinate system before you draw. This coordinate system is a transformation of the base coordinate system and has a more convenient origin, at the lower left corner of the area you’re drawing in. Figure 4-5 illustrates the relationship between the base coordinate system and the default coordinate system provided for a View.

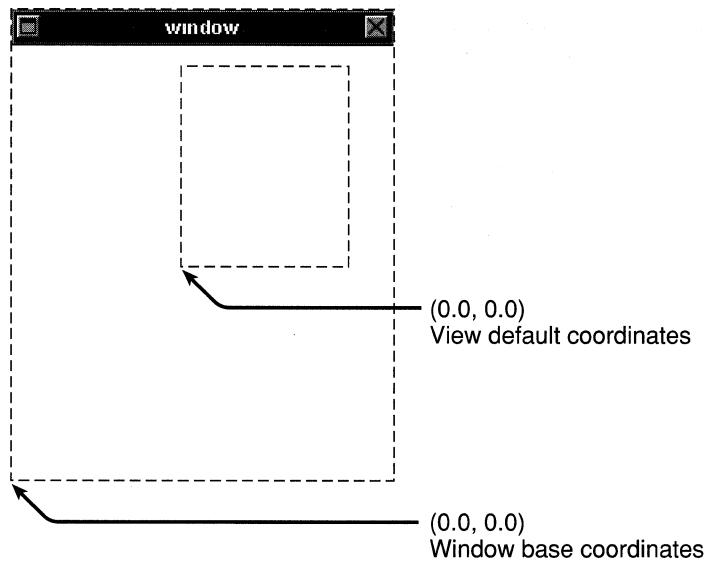


Figure 4-5. Window and View Coordinates

You can use Application Kit methods or PostScript operators, or both, to further modify these default coordinates. See Chapters 6 and 7 for more on Views.

Borders and Content Areas

The initial size of a window is determined by parameters that specify its width (along the x-axis) and its height (along the y-axis) in the screen coordinate system. These two parameters define the window's initial *content area*, the area that's available for drawing. The window's initial location is set by parameters that specify the lower left corner of the content area.

Windows typically have a title bar above the content area and a border around both the content area and title bar. Often they also have a resize bar below the content area and inside the border. (Some windows may have a border but lack a title bar and resize bar; some lack even the border.)

The window border, title bar, and resize bar lie outside the requested window area. If you ask the Application Kit to create a titled window 100 pixels wide and 100 pixels high, the window border and title bar will surround the 100-by-100 square you asked for. The point where you locate the window will correspond to the lower left corner of the square, inside the border. The border, title bar, and content area all lie inside the window, and share the window's coordinate system.

Figure 4-6 is a close-up view of the corner of a titled window created at (50.0, 300.0) in the screen coordinate system. The window border is just one pixel wide. The point where the window was located becomes the origin of the initial drawing coordinates within the

content area (the coordinate origin for a View that fills the content area). The origin of the base coordinate system lies at the lower left corner of the window itself, outside the border.

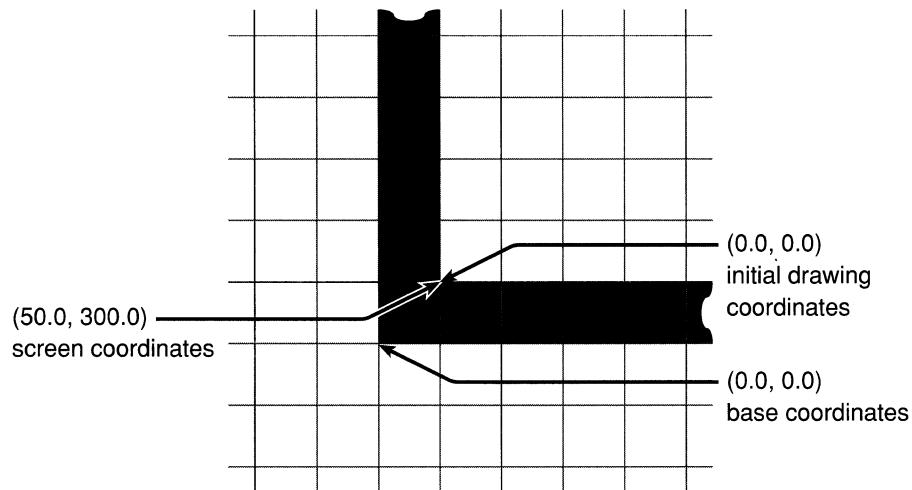


Figure 4-6. Window Corner

Printing Coordinates

When a screen image is printed by the NeXT 400 dpi Laser Printer, or by any other PostScript printer, one unit within the base or screen coordinate system (that is, one pixel) translates to 1/72 inch, equal to about one typographical point. Since a screen pixel is smaller, approximately 1/92 inch, the screen shows images at just over three-fourths their printed size. This reduction permits a full 8 1/2-by-11-inch page to be shown on the screen from top to bottom, without scaling the window's coordinate system.

This doesn't mean that an image must always be shown on the screen at three-fourths its printed size. It's possible, for example, to use the PostScript **scale** operator to "magnify" the image on the screen, then print the image without the scaled magnification. The screen version could actually be larger than the unmagnified printed version.

Laser printers produce images with many more pixels per inch than do screen displays. The NeXT laser printer prints either 300 or 400 pixels per inch, depending on the setting. Printed images, therefore, have a higher resolution and are portrayed in more detail than screen images.

For example, a vertical line with a specified width of one unit within the base coordinate system can be shown on the screen as a thin rectangle just one pixel wide. When the line is printed, it will look almost 28% larger and have a width of at least four printer pixels. Setting the width of this line to less than 1—to 0.25, for example—will have no effect on the dimensions of the line on-screen, but will make the line thinner when it's printed.

Rectangles

It's often necessary to limit a PostScript operation, or drawing in general, to a particular area of the screen. These areas are most often specified as rectangles:

- A window is a rectangle in the screen coordinate system.
- Several PostScript operators, such as the compositing operators discussed under “Compositing and Transparency” below, act on rectangles within windows.
- The Application Kit uses rectangles extensively to designate particular areas—called “Views”—where drawing can occur inside windows (see Chapter 6 for details).

On the NeXT computer, a rectangle is a combination of a point and an extent. The point locates the rectangle by assigning it x and y values within a coordinate system. The extent specifies the size of the rectangle—its width as measured in a direction parallel to the x-axis and its height as measured in a direction parallel to the y-axis. The width and height are measured from the point that locates the rectangle.

This way of specifying a rectangle has one consequence of note: A rectangle is aligned with its coordinate system; the sides of the rectangle are parallel to the coordinate axes. Figure 4-7 illustrates a rectangle with the orientation of the base or screen coordinate system.

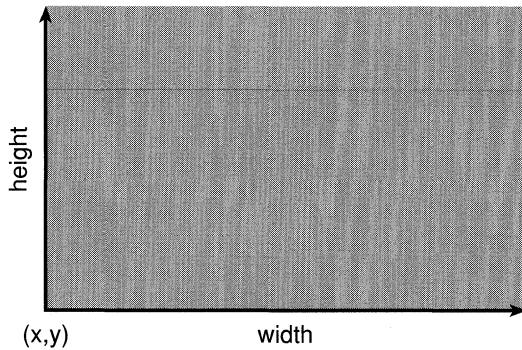


Figure 4-7. Rectangle

PostScript operators take four parameters to specify a rectangle. An example is **rectfill**, which fills a rectangle with the current color:

```
x y width height rectfill
```

In the Application Kit, rectangles are specified as C structures of type NXRect. The NeXT header file **event.h** (in the **dpsclient** subdirectory of **/usr/include**) defines the elements that make up an NXRect structure:

- Since all coordinate values must be specified by floating-point numbers, it defines NXCoord as a **float**:

```
typedef float NXCoord;
```

- A pair of NXCoord variables—one for the x coordinate and one for the y coordinate—designate a point:

```
typedef struct _NXPoint {
    NXCoord x;
    NXCoord y;
} NXPoint;
```

- A pair of NXCoord variables also designate the size of a rectangle:

```
typedef struct _NXSize {
    NXCoord width;
    NXCoord height;
} NXSize;
```

The NeXT header file **graphics.h** (in the **appkit** subdirectory) combines NXPoint and NXSize structures to define the rectangle itself:

```
typedef struct _NXRect {
    NXPoint origin;
    NXSize size;
} NXRect;
```

The values in an NXSize structure should never be negative; for the Application Kit, the width and height of a rectangle must be positive (or 0.0). This means that the point that locates a rectangle (**origin**) will have the smallest coordinate values of any point in the rectangle. The extent of a rectangle is measured in positive directions from its origin.

Rectangle Geometry

From a pair of rectangles within the same coordinate system, it's possible to calculate a third rectangle, their *union*, the smallest rectangle that completely encloses them both.

It's often also possible to calculate a fourth rectangle, their *intersection*. Two rectangles are said to intersect if they have any area in common. Since rectangles within the same coordinate system have parallel sides, this area will also be a rectangle.

Rectangular union and intersection are illustrated in Figure 4-8.

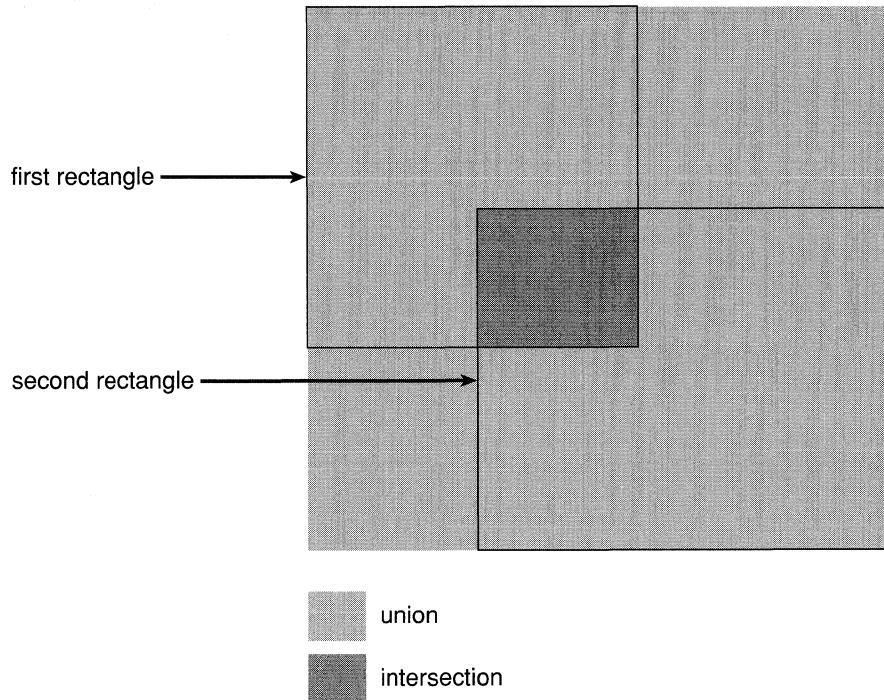


Figure 4-8. Union and Intersection of Rectangles

When passed pointers to two NXRect structures, **NXIntersectRect()** returns YES if the two rectangles intersect and NO if they don't.

```
BOOL overlap;
overlap = NXIntersectRect(&rect1, &rect2);
```

If the two rectangles intersect, you can use **NXIntersectionRect()** to calculate the area that overlaps. It will place the intersection in the structure specified by its second argument and return a pointer to the same structure.

```
NXRect *smallrect;
smallrect = NXIntersectionRect(&rect1, &rect2);
```

If the two rectangles don't overlap, **NXIntersectionRect()** returns a NULL pointer and doesn't modify either of the rectangles it's passed.

NXUnionRect() is similar to **NXIntersectionRect()**. It places the union of the two rectangles in the structure specified by its second argument and returns a pointer to the structure.

```
NXRect *bigrect;
bigrect = NXUnionRect(&rect1, &rect2);
```

NXUnionRect() and **NXIntersectionRect()** perform two of the most common calculations on rectangles. But there are also a number of other functions that modify NXRect structures in prescribed ways:

- **NXSetRect()** initializes an NXRect structure.
- **NXIntegralRect()** alters a given rectangle so that it has no fractional coordinates. If the coordinate system hasn't been scaled or rotated, it guarantees that the sides of the rectangle will lie on pixel boundaries.

NXDivideRect() slices a rectangle in two. You can specify the size of the slice and the side from which it should be taken.

- **NXInsetRect()** calculates a rectangle that's inset from the one given.
- **NXOffsetRect()** moves a rectangle by specified offsets along the x- and y-axes.

Another function, **NXPointInRect()**, is used to determine whether a particular point (such as the location of the cursor) lies within a particular rectangle.

```
BOOL inside;
inside = NXPointInRect (&point, &rect);
```

It returns YES if the point touches the rectangle, and NO if it doesn't.

All these functions are defined in the Application Kit. They're described in more detail in the *NeXTstep Reference* manuals.

When a Rectangle Isn't Rectangular

The sides of a rectangle are always parallel to the x- and y-axes of its coordinate system. Since the coordinate axes are typically perpendicular to each other, the corners of a rectangle are, as expected, 90° angles.

It's possible, however, to modify the PostScript coordinate system so that the axes intersect at an angle other than 90°. A rectangle specified within such a coordinate system won't appear to be a rectangle at all. It will have the shape of a nonrectangular parallelogram; since its sides are parallel to the coordinate axes, they won't meet to form 90° corners. This is illustrated in Figure 4-9.

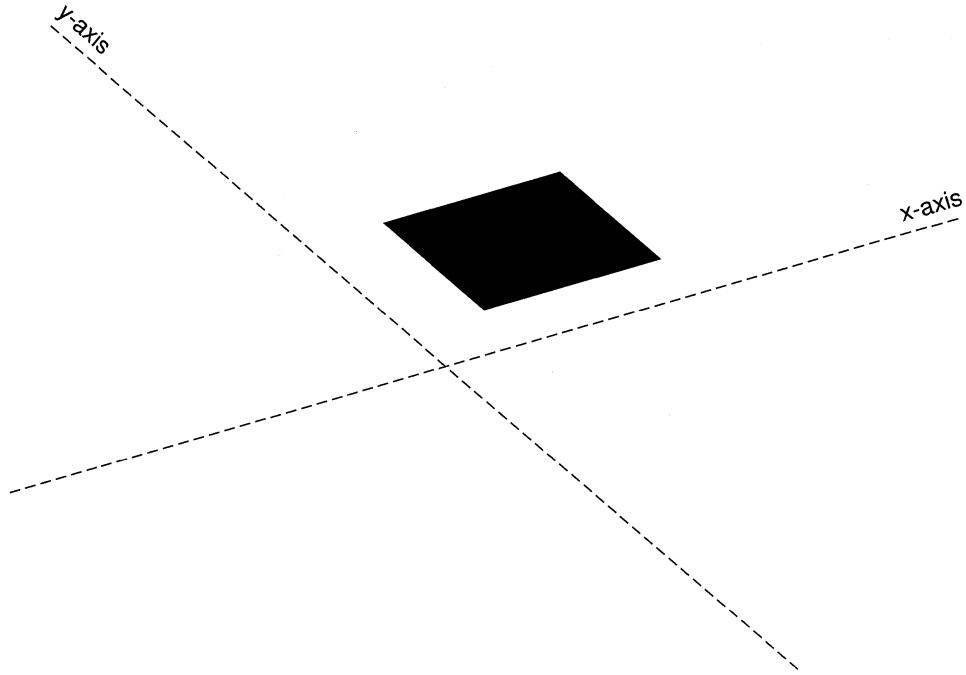


Figure 4-9. Nonrectangular Rectangle

The most straightforward way to produce a skewed coordinate system is with the following sequence of two PostScript transformations:

1. Scale the coordinate system unequally.
2. Rotate the coordinate axes into the scaled coordinates.

For example, Figure 4-9 above shows the approximate result of this PostScript code:

```
2.0 1.0 scale
30.0 rotate
40.0 40.0 100.0 100.0 rectfill
```

Here the **rectfill** operator produces a rectangle at (40.0, 40.0) with a width and height both equal to 100.0. In an unscaled, unrotated coordinate system, it would appear as an upright square.

Before the coordinate transformations in this example, the x-axis was horizontal and the y-axis was vertical; units along both axes had the same length. The **scale** operator stretched the length of a horizontal unit (measured along the x-axis) to twice the length of a vertical unit (measured along the y-axis). The **rotate** operator then altered the position of the x- and y-axes, but not the horizontal and vertical orientation of the scaling. Like anything else located in this scaled coordinate system, the x- and y-axes are stretched in a horizontal direction, and are thus pulled away from the perpendicular. Figure 4-10 illustrates what happened to the x-axis in the previous figure when it was rotated 30 in the scaled coordinate system.

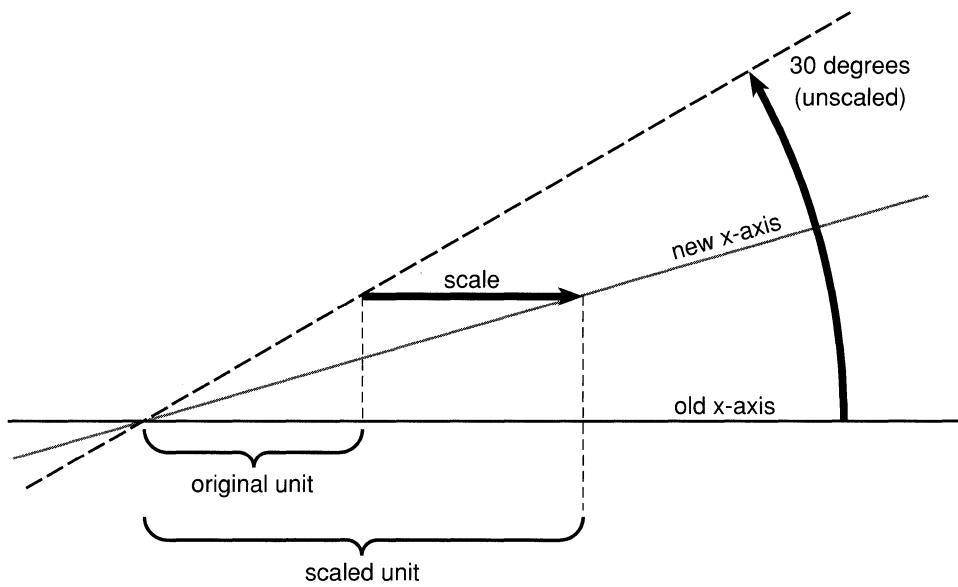


Figure 4-10. Rotated x-Axis in Scaled Coordinates

Although skewed coordinates turn rectangles into parallelograms, NeXT documentation refers to any area specified by a point and an extent as a “rectangle,” no matter what shape it has. When viewed from within a skewed coordinate system, a nonrectangular rectangle still has 90° angles; it’s nonrectangular only when viewed from the outside.

The Window System

The NeXT window system is grounded in extensions to the Display PostScript system. The extensions create windows, move and resize them, order them from front to back, direct drawing to them, associate events with them, remove them from the screen, and carry out other low-level window management tasks.

The Application Kit builds on these operators and provides an object-oriented programming interface to them. An application should create the windows it needs through the Kit. The Kit defines Window objects to manage windows, link them to an event-handling mechanism, and oversee the drawing that’s done in them. Creating a Window object through the Kit produces a window to draw in and provides a structure that integrates the window into the application. The methods for creating and managing windows are discussed under “Managing Windows” in Chapter 6.

Window Numbers

Applications typically use a number of different Window objects and keep track of them through their object **ids**. The Window Server keeps track of windows by assigning each one a unique *window number* as an identifier. The window number is an **int** guaranteed to be greater than 0.

No two windows, even if they belong to different applications, will be assigned the same window number. However, when a window is destroyed, the Window Server may reuse its number.

Contexts and Graphics States

The Window Server can serve a large number of client applications; its PostScript interpreter interprets PostScript code concurrently for all running applications.

For each connection that it has to an application, the Window Server maintains an independent PostScript execution context. Figure 4-11 illustrates the context that each application sees.

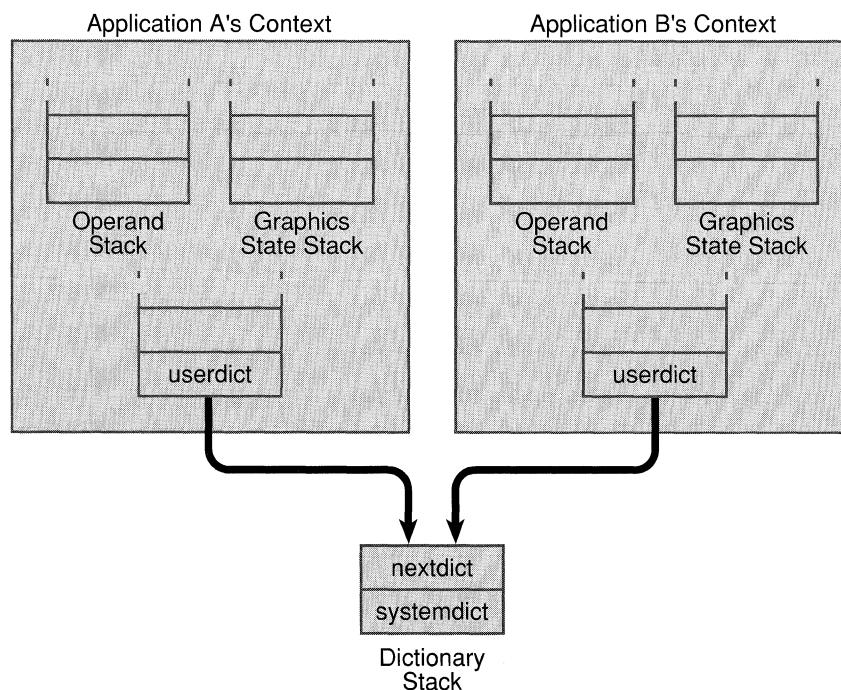


Figure 4-11. PostScript Execution Contexts

Every context has its own set of stacks, including an operand stack, graphics state stack, and dictionary stack. There are three built-in dictionaries in the dictionary stack. From top to bottom, they are **userdict**, **nextdict**, and **systemdict**. **userdict** is private to the context, while **nextdict** and **systemdict** are shared by all contexts. **nextdict** is a modifiable dictionary containing any dynamic information common to all applications, such as downloaded packages. **systemdict** is a read-only dictionary containing all the PostScript operators, both the standard ones and those implemented by NeXT.

The Current Window

Each context has a current graphics state and graphics state stack. The current graphics state is a set of parameters that the PostScript interpreter uses when rendering images on the screen (or printed page). It includes parameters for such things as the current color, line width, clipping path, and dash pattern.

Three parameters are especially important to the window system:

- The current device, where drawing is to be rendered
- The current transformation matrix (CTM), which maps the coordinate system the application is currently using to the device
- The current clipping path, which defines the area where images can be rendered

When drawing on the screen, the device is a window. The device of the current graphics state is the *current window*. PostScript painting operators, such as **stroke**, **fill**, and **show**, draw directly in the current window. You don't need the **showpage** or **copypage** operators to make images visible.

When the current graphics state is set to a new window device, the other two parameters also change:

- The current transformation matrix is initialized to the window's base coordinate system. (Thereafter, the **initmatrix** operator can be used to reestablishes this coordinate system.)
- The current clipping path is set to a path around the window. (The **initclip** operator reestablishes this clipping path.)

These changes ensure that drawing is confined to the current window and is in a coordinate system specific to the window.

Changing Graphics States

As users shift their attention from place to place on the screen, applications are required to shift the drawing focus from window to window and graphics state to graphics state. There can be only one current graphics state, and therefore only one current window, for each execution context.

The graphics state stack saves former graphics states that might later be restored. The **gsave** operator pushes the current graphics state on the stack and **grestore** replaces the current graphics state with one from the stack. This is illustrated in Figure 4-12.

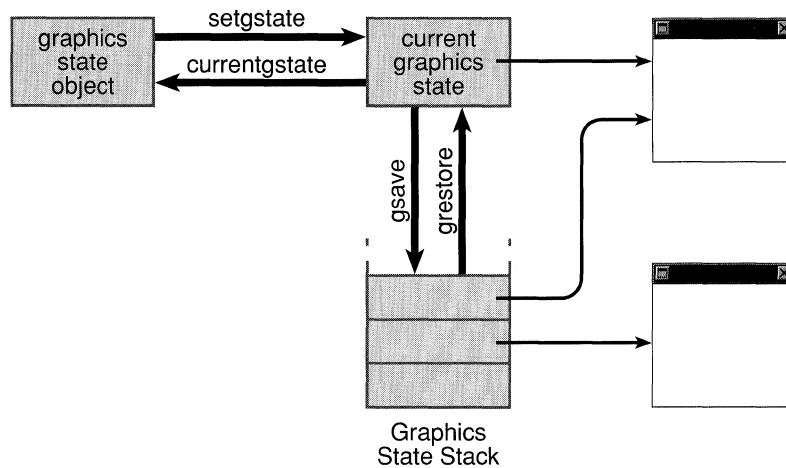


Figure 4-12. Graphics States

Because applications typically draw in many windows, it's possible for each graphics state on the stack to point to a different window device.

The stack saves graphics states in a particular order, but as users shift their attention on-screen, an application can be required to switch between graphics states in an order that the stack can't predict. Therefore, to simplify the task of switching between graphics states, the Display PostScript system permits graphic states to be saved not only on the stack, but also in individual graphics state objects. The **gstate** operator creates a new graphics state object and copies the current graphics state to it. The code below saves the current graphics state in an object named **foo**:

```
/foo gstate def
```

A graphics state object is not stored on the stack; the **gstate** operator allocates memory for it. Applications therefore tend not to create many graphics state objects for themselves, but reuse the ones they do create.

The **currentstate** operator copies the current graphics state to a graphics state object, and the **setstate** operator copies the graphics state recorded in a graphics state object to the current graphics state. These two operators are illustrated in Figure 4-12 above. They serve functions parallel to **gsave** and **grestore**.

At minimum, the Application Kit keeps one graphics state object for each window. The object identifies the window (as the device) and records its base coordinate system. Shifting the drawing focus from one window to another is a matter of first setting the current graphics state from the desired window's graphics state object and then altering parameters of the current graphics state as needed. Altering these parameters doesn't affect the graphics state object.

Additional graphics state objects can be assigned to facilitate shifting back and forth between particular graphics states within a window. For example, one object could be assigned to a text display and another to the scroller that scrolls the display. Views created through the Application Kit should be assigned graphic state objects using Kit methods rather than PostScript operators.

Window Buffering

Windows often overlap when there's more than one on the screen. When a group of windows are stacked together, a mouse click within one of them usually brings it to the front and its previously obscured contents become visible. Whether the contents are automatically made visible by the Window Server or must be redrawn by the application depends on whether the window has a backup buffer. The buffer stores pixel values for the portions of a window that aren't shown on the screen. When those portions become visible again, they can be copied from the buffer by the Window Server rather than be redrawn by the application.

So that you can pick the buffering scheme that's best for the needs of your application, three distinct types are provided:

- *Nonretained*. A nonretained window has no buffer. All drawing is done directly on-screen—that is, into the memory dedicated to holding screen pixel values. If part of the window gets covered by another window, the memory for that part of the screen changes and a portion of the covered window's contents are lost. If later the window is uncovered, any drawing in that part will be replaced by the background color.
- *Retained*. A retained window has a buffer equal to the size of the window. Drawing is done directly on-screen—directly into screen memory—for all visible portions of the window, and into the buffer for all portions of the window that are covered and not visible. If a visible portion of the window becomes obscured, the contents of that portion are saved in the buffer. If the obscured portion of the window is later revealed, the contents of the buffer are copied on-screen.
- *Buffered*. Like a retained window, a buffered window has a buffer large enough to hold its entire contents. However, the buffer isn't only backup storage, it also serves as a true input buffer. All drawing is first done in the buffer, then is copied to the screen through an explicit call to the NeXT PostScript **flushgraphics** operator. Drawing done through the Application Kit's display methods is automatically flushed.

These three types are illustrated in Figure 4-13.

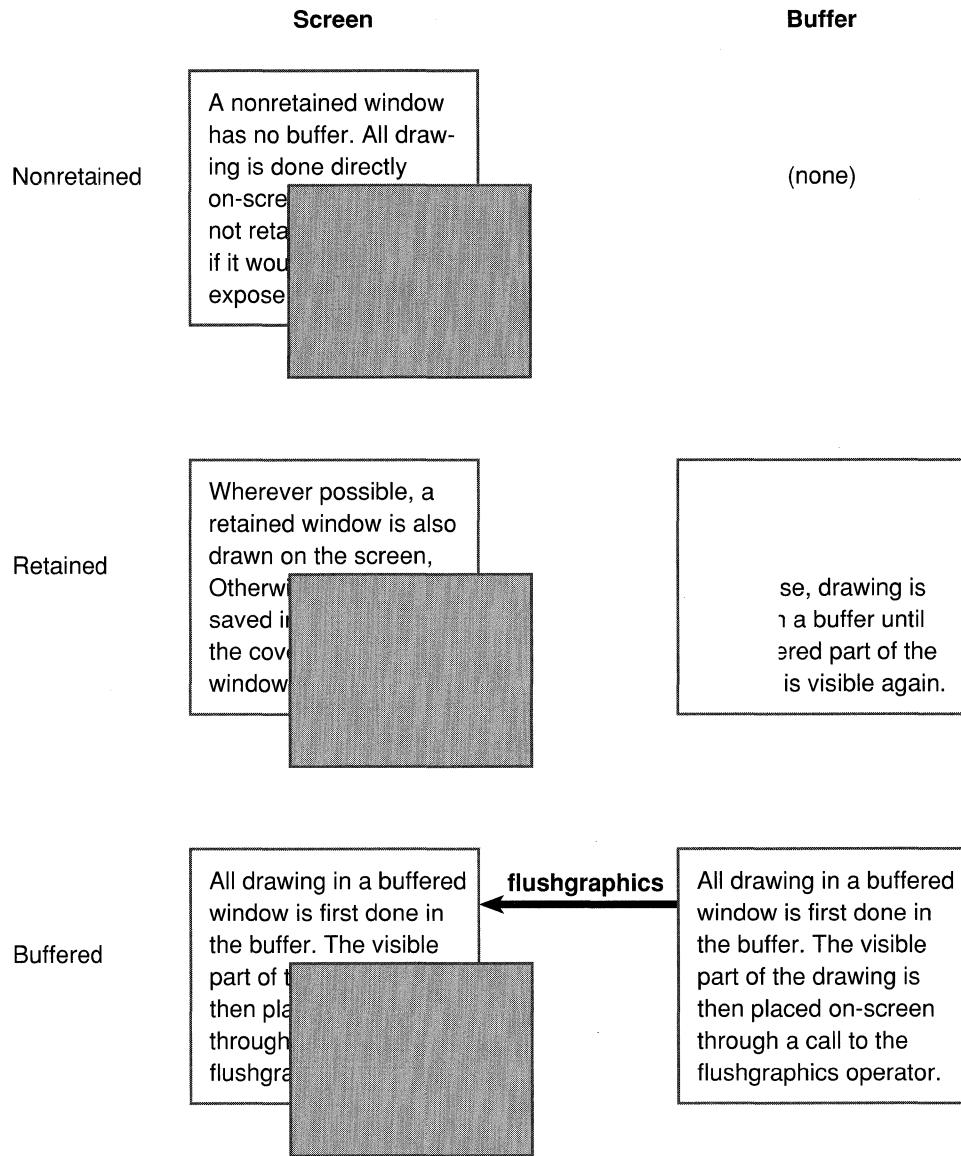


Figure 4-13. Window Buffering

When a covered portion of a retained or buffered window becomes visible, the newly uncovered portion is automatically refreshed from the buffer. However, an uncovered portion of a nonretained window needs to be redrawn from scratch. Nonretained windows that are created through the Application Kit can handle this automatically. See “Kit-Defined Events” in the “Event Handling” section of Chapter 7.

The NeXT header file **dpsNeXT.h** (in **/usr/include/dpsclient**) defines three C constants that you can use to specify the buffering type when creating a window through Application Kit methods:

```
NX_NONRETAINED  
NX_RETAINED  
NX_BUFFERED
```

When created directly with PostScript operators, the equivalent constants are:

```
Nonretained  
Retained  
Buffered
```

Choosing a Buffering Type

Because it combines the simplicity of drawing directly on the screen with the security of a backup buffer, a retained window is usually preferred over a nonretained window. Nonretained windows are appropriate for transitory images that you don't need to save, or for windows you can guarantee will never be covered. However, in a multi-application environment, this guarantee is a hard one to give.

A buffered window is appropriate when you're drawing images that take some time to appear, and you don't want users to watch them being rendered on-screen. With a buffered window, an elaborate picture can be displayed first in the buffer, and when it's complete be flushed instantaneously to the screen.

Buffered windows also yield smoother transitions between images when you're repeatedly drawing in the same area of the window. Suppose, for example, that your application draws the word "RUNNING" and then replaces it with "FINISHED". PostScript operators paint only within the outline of images they're rendering—in this case, within the outlines of the letters of the two words. Therefore, it's necessary to erase "RUNNING" before drawing "FINISHED"; otherwise, some of "RUNNING" will show through the spaces between the letters of "FINISHED". PostScript code like the following might be used:

```
1 setgray                      % set the current color to white
0 0 120 20 rectfill            % erase
0 setgray                      % set the current color to black
0 0 moveto
(FINISHED) show                % show "FINISHED"
flushgraphics                  % flush, if it's a buffered window
```

The Display PostScript operator **rectfill** first erases a rectangular area by filling it with white, then **show** draws "FINISHED" in black.

Since a buffered window erases and redraws in the buffer, then flushes the buffer to the screen, you won't see the first image ("RUNNING") removed before the second image ("FINISHED") replaces it. However, a retained window erases and redraws on-screen. This difference doesn't much matter for a case where one word replaces another. But where images rapidly replace each other—as in an animation or redrawing lines of text in response to the user's typing—you'll notice a slight flicker in a retained window as images are erased before they're replaced.

The Screen List

The Window Server keeps track of the current front-to-back order of windows through a *screen list*. The list is rearranged every time the user brings a window to the front or sends one to the back. Applications reorder windows on-screen by reordering them in the list. This should be done through Application Kit methods. See Chapter 6 for details.

A window can be placed at the top of the screen list—that is, in the frontmost position—at the bottom of the list, or in any position above or below another window. Every window has a unique position in the list; even if two windows don't overlap on the screen, one of them is listed in front of the other. Windows can also be left off the list, and thus kept off the screen.

When you position a window in the list, you must specify two parameters:

- Whether it's to be placed above or below another window, or left off the list entirely.
- What the window number of the other window is. If this number is 0, the window you're ordering will be placed at the very top or bottom of the list. If the window you're ordering is to be left off the list, this second parameter is ignored.

The NeXT header file **dpsNeXT.h** (in **/usr/include/dpsclient**) defines three C constants that can be used to order windows:

```
NX_ABOVE  
NX_BELOW  
NX_OUT
```

In PostScript code, the constants are:

```
Above  
Below  
Out
```

Window Tiers

The Application Kit divides the screen list into six tiers. The top five tiers keep special types of windows from being buried under other windows when they're on-screen. The tiers are:

1. Pop-up and pull-down lists
2. Attention panels
3. The main menu and attached submenus
4. Other menus
5. Docked icons
6. All other windows

See Chapter 2, “The NeXT User Interface” for more on window tiers, and Chapter 6 for information on how to keep windows in the proper tier.

Off-Screen Windows

Windows are created off-screen. Applications must place them in the screen list to make them visible. If a window is buffered or retained, it’s a good idea to draw in it before moving it on-screen. This prevents the user from seeing images rendered in stages. Windows that are kept permanently off-screen can be useful for caching images that you want to composite to on-screen windows. (See “Compositing and Transparency” later in this chapter for more information on compositing from an off-screen window.)

An off-screen window is entirely invisible; it has no screen memory. All drawing goes directly into the window’s buffer. Drawing in an off-screen nonretained window is pointless; since it has no buffer, there’s nowhere for the drawing to go. Flushing an off-screen buffered window is similarly pointless.

You can put a window in the screen list, and take it out again, with Application Kit methods (described under “Managing Windows” in Chapter 6). A window disappears from the screen when it’s removed from the list, and is restored to the screen when it’s put back in the list. (A removed window disappears only because covered portions of the windows behind it—including the background window for the entire screen, **workspaceWindow**—are redisplayed.)

It’s also possible to locate windows off-screen yet keep them in the screen list. Because the screen coordinate system extends far beyond the screen along both the x- and the y-axis, you can assign coordinates that prevent a window from being visible. An ordinary-sized window located at $(-5000.0, -5000.0)$ would not show up on the screen, for example.

For two reasons, it’s preferable to keep windows off-screen by removing them from the screen list rather than by assigning them off-screen coordinates:

- Events—user actions on the keyboard and mouse—can be associated with any window in the screen list, whether its coordinates place it on the screen or off. However, windows not in the list don’t get events. Since events are usually associated with an on-screen display (a slider or button, for example) and require visual feedback, they shouldn’t normally be associated with windows the user can’t see. Events are discussed in the next chapter.
- Windows that aren’t in the screen list can be assigned on-screen coordinates. This simplifies subsequent handling; placing the window on-screen involves no coordinate adjustments and no effort to remember where the window was located.

The Background Color

Windows created through the Application Kit should be assigned a background color with methods defined in the Kit. This section describes the background color of windows at a lower level, the level of PostScript operators.

When a new window is created, the pixels within it are initialized to a *background color*, usually white. You can choose a color other than white by combining the effects of a few PostScript operators.

Through the **setexposurecolor** operator, you can determine the color that's displayed when new areas of the window are exposed. This operator sets the color that's shown in areas added to a window when it's resized larger. It also sets the color that's used when covered areas of a nonretained window are exposed. (When retained and buffered windows are exposed, their on-screen appearance is determined by the contents of their buffers.)

This code sets the background color of the current window to 50% gray:

```
0.5 setgray           % Set a new gray value
setexposurecolor      % Set the background color to new gray value
```

To make the initial color of a nonretained window match the color set with this operator, create the window off-screen, set the background color, then move the window on-screen.

You can set the background color of a retained or buffered window by a combination of the **fill** operator (for the initial surface area of the window) and the **setexposurecolor** operator (for any future surface area if the window is resized). The window should be created off-screen and its background color set, before being moved on-screen.

When an area of a window is erased, you may want the color shown to be the same as the background color. Since the **erasepage** operator erases to white, don't use it unless white is the color you want. Instead make the clipping path the current path and fill it with the background color of your choice. This code erases to 50% gray:

```
clippath            % Make the clipping path the current path
0.5 setgray         % Set a new gray value
fill                % Fill current path with new gray value
```

Compositing and Transparency

Compositing is a NeXT extension to the Display PostScript system that enables separately rendered images to be combined into a final image. It encompasses a wide range of imaging capabilities:

- It provides the means for simply copying an image “as is” from one place to another.
- It lets you add two images together so that both appear in the composite superimposed on each other.
- It defines a number of operations that take advantage of *transparency* in one or both of the images that are combined. When they’re composited, the transparency of one image can let parts of the other show through.

Compositing can be used for copying within the same window on the screen, as during scrolling, or for taking an image rendered in one window and transferring it to another. Images are often stored in off-screen windows and composited into windows on-screen as they’re needed. For example, the Workspace Manager first renders application and document icons in off-screen windows, then copies them on-screen.

When images are partially transparent, they can be composited so that transparent sections of one determine what you’ll see of the other. Each compositing operation uses transparency in a different way. In a typical operation, one image provides a background or foreground for the other. When parts of an image are transparent, it can be composited over an opaque background, which will show through the transparent “holes” in the image on top. In other operations, transparent sections of one image can be used to “erase” matching sections of the image it’s composited with. In most operations, the composite is calculated from the transparency of both images.

Compositing with transparency can achieve a variety of interesting visual effects. A partially transparent, uniformly gray area can be used like a pale wash to darken the image it’s composited with. Patches of partially transparent gray can add shadows to another image. Repeated compositing while slowly altering the transparency of two images can dissolve one into the other. Or an animated figure can be composited over a fixed background.

Before images can be composited, they must be rendered. To take advantage of transparency when compositing, at least one of the images needs to be rendered, to some extent, with transparent paint.

The next section describes how to create transparent images for compositing. If you’re interested only in using compositing to copy, turn to the later section titled “Compositing.”

Transparent Paint

On the NeXT computer, you can set a coverage parameter in the current graphics state in addition to the color parameter:

- The **setalpha** operator sets the current coverage to an *alpha value* in a range from 0 (completely transparent) to 1 (completely opaque).

```
0.333 setalpha
```

The default alpha value is 1. To use transparency, you must explicitly set a lower value; **currentalpha** returns the current setting.

- The **setgray** operator sets the current color to a value in a range from 0 (black) to 1 (white):

```
0.5 setgray
```

currentgray returns the current setting.

Note: In this discussion, the color parameter is illustrated by the single value, sometimes referred to as a *gray value*, that's set by **setgray** and returned by **currentgray**. However, the principles of compositing and transparency outlined here also apply when the color parameter has more than one component. For example, the red, green, and blue components in an RGB color system can each be set to a different value, but each would interact with the alpha value just as the gray value does.

Together, the alpha value and color value determine the kind of paint that's used when rendering an image. The color value determines the color of the paint; the alpha value determines how opaque or transparent it is. In the example above, the paint is mostly transparent; only one third of it is opaque. If it had been made completely transparent, as in

```
0.0 setalpha
```

the current color would have been irrelevant; images drawn with transparent paint are invisible.

Figure 4-14 below illustrates the kind of paint that would be produced from an alpha value of 0.333 and a color value of 0.5. Setting the color value midway between white and black is like mixing equal equal portions of white and black paint (or, on the NeXT MegaPixel Display, equal portions of light gray and dark gray) to produce a medium gray. Making this paint only a third opaque is like mixing one portion of it with two portions of transparent paint.

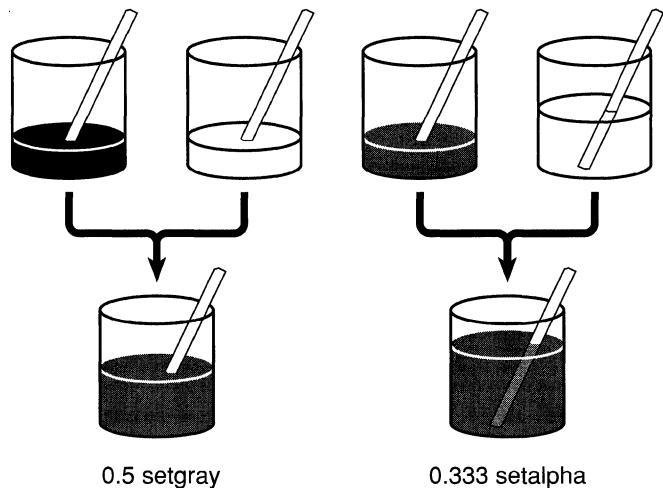


Figure 4-14. Mixing Transparent Paint

The color is still 0.5, midway between black and white, but the paint is diluted with transparency. When drawing with this paint, as in

```
0 0 100 100 rectfill
```

some of the background will show through.

What paint with a gray value of 0.5 and an alpha value of 0.333 actually looks like depends on the background it's applied against. The paint is just one-third opaque, so it contributes only one-third of the visual result. The background contributes the other two-thirds.

The background that shows through transparent paint is device-dependent. You won't see the images already displayed in the area being painted; they'll be erased no matter what paint you use. Rather, the background depends solely on the imaging model assumed by the device. There are two principal, and competing, models:

- In the PostScript imaging model, an image is built up by applying paint on a white surface. This is a convenient model for printers, which normally put ink on white pages, and it fits well with the assumptions of artists who begin with a white canvas.
- In the competing model, images are built up by adding color to a colorless, black surface. This model is commonly assumed in video and color graphics.

Devices can also assume backgrounds other than black or white, but black and white are the only two that need to be discussed here.

Figure 4-15 below shows how each of these imaging models would affect the partially transparent gray paint in the example above.

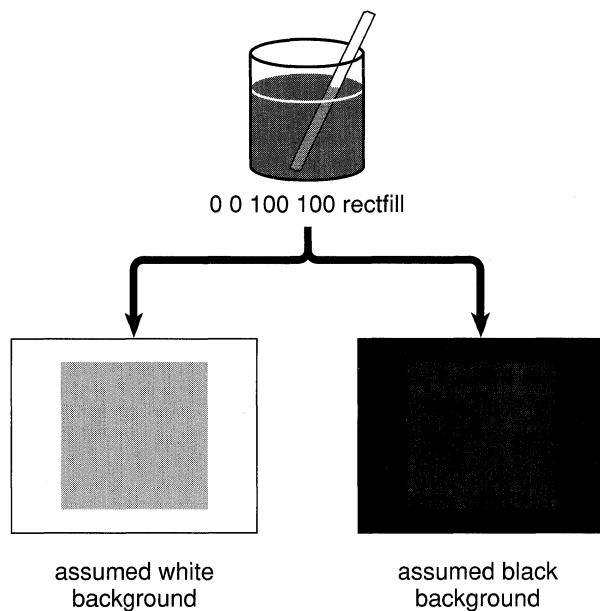


Figure 4-15. Painting with Transparency

If the background color is black, the result will appear blacker than the medium gray we originally set. If the background color is white, the result will be whiter.

The NeXT MegaPixel Display adopts the PostScript model: A white background is assumed. Areas painted with completely transparent paint appear white on-screen; areas painted with partially transparent grays appear as lighter grays. However, for external data representation, the second imaging model, with a black background, is assumed. See “Data Representation” below for details.

In either case, the appearance of the painted image may be deceptive, since, when composited, the transparency of the image may let some of the other image (rather than a black or white background) show through.

Note that painting destroys any images that were previously rendered in the same area. Painting one image after another completely obliterates the first image. Even when the paint is totally transparent, the images it covers are destroyed (in favor of the background color assumed by the device). The only way to combine images is through compositing.

Data Representation

Painting operators (such as **rectfill** and **stroke**) produce an image on-screen or in a window buffer. They directly affect pixel values.

The values that painting operators produce depend on a number of factors, including:

- The alpha and color values of the current graphics state.
- The transfer function (which maps values set by **setgray** and **setalpha** to values used by the device).
- The current halftone screen. (See “Pixels, Halftones, and Rectangular Coordinates” above for information on halftones.)

The range of values that a pixel can represent depends on the number of bits provided per pixel. On the MegaPixel Display, two bits of memory are set aside to store the color of each pixel, and another two to store its coverage. Values for the coverage of pixels are stored separately from color values.

Every pixel within a window has both color and coverage components. However, for efficiency, memory isn’t allocated for the coverage component if all the window’s pixels are opaque. Instead of being explicitly stored, the opaqueness of the window is implicit. Once any pixel needs to be assigned an explicit coverage value (because it’s no longer completely opaque), alpha storage is allocated for the entire window.

There are significant performance advantages associated with windows that have implicitly opaque pixels. Imaging in such a window is twice as fast as imaging in a window with explicit coverage, since only half as much memory is affected. Copying between implicitly opaque windows is optimized to be highly efficient; compositing operations between windows with explicitly stored coverage values can be several times slower.

The **currentwindowalpha** operator, described in the *NeXTstep Reference, Volume 2*, reports whether a window has an explicit coverage component. It’s useful mainly for debugging; you’d rarely need to know how a window’s coverage values are stored.

Bitmaps

A *bitmap* is binary data that describes the pixel values for an image. It’s an external data representation, perhaps kept in a file or in memory allocated by your program, as opposed to the representations used internally by the device.

A bitmap is the most efficient way of rendering a complicated image in a window; the external data values in the bitmap can be translated directly into device-internal pixel values. The **NXImageBitmap()** function displays a bitmap image, and **NXReadBitmap()** produces a bitmap from an existing image. (These two functions are described in the *NeXTstep Reference, Volume 2*.)

Bitmaps cover a rectangular area and describe all the pixels within the rectangle. Because the MegaPixel Display requires two bits of memory for each component of a pixel, a bitmap also assigns two bits per pixel for each component. Bitmaps always have a color component and may or may not also have a coverage component. If all the pixels described by the bitmap are completely opaque, the coverage component can be omitted.

By convention, bitmaps use the same color and coverage scales as `setgray` and `setalpha`:

Color	Value	Coverage
White	1	Opaque
Black	0	Transparent

Where there's more than one bit of storage per pixel, all the bits for a component must be "on" for the pixel to be white or opaque, and all must be "off" for it to be black or transparent. Because there are two bits per pixel for both color and coverage on the MegaPixel Display, each pixel can have any of four values in each component:

Color	Coverage
White	Opaque
Light gray	2/3 opaque
Dark gray	1/3 opaque
Black	Transparent

Not every combination of color and coverage values is permitted, however. When there's any degree of transparency, color values are adjusted toward black. This adjustment is explained in the next section.

Premultiplication

It's convenient, both in bitmaps and in device-internal image representations, to store color values that reflect the effect of the coverage component. The intended or "true" color (as determined by `setgray`) is multiplied by the coverage before being stored. So a white pixel (true color = 1) that's one-third transparent (coverage = 0.667) will store a color value of 0.667 (1×0.667).

There are two reasons for this adjustment:

- It's a required part of all compositing operations. Doing the multiplication when pixel values are first stored—"premultipling"—means that it can be avoided during compositing. This makes compositing more efficient.
- It results in a value that more accurately reflects the pixel's actual color contribution.

For example, when the color parameter is set to 0.5 and the coverage parameter equals 0.333, as illustrated in Figure 4-14 above, color values are premultiplied to 0.167 (0.5×0.333). This value reflects the visual result of placing this mostly transparent paint over an assumed black background, as shown in Figure 4-15 above; the background biases the color toward black. If 0.167 isn't one of the pure colors available on the device, the actual color values for pixels in the painted area will be determined by the halftone pattern for 0.167 gray.

Where pixel values are opaque (coverage = 1), premultiplication doesn't change color values. But where there's any degree of transparency, premultiplying results in color values that are blacker than the true values set. This reflects the effect of an assumed black background showing through. In bitmaps, color values must be premultiplied "toward black" in this way. This means that, in external data representations, a pixel's color value can never be greater than its coverage value. The table in Figure 4-16 below summarizes permitted color and coverage combinations.

Color Coverage \	Black	Dark Gray	Light Gray	White
Opaque	●	●	●	●
2/3 Opaque	●	●	●	
1/3 Opaque	●	●		
Transparent	●			

Figure 4-16. Permitted Pixel Values

Note that the color of a completely transparent pixel is black, and a white pixel can only be opaque. Only opaque pixels can show the full range of colors.

Although a black background is assumed for the external data representation stored in bitmaps, the internal representations on the MegaPixel Display use the opposite convention. Internally, color values are premultiplied towards white rather than black. Where there's transparency, stored color values are whiter than the true color, reflecting an assumed white background showing through. **NXImageBitmap()** and the imaging operators on which it's based accurately translate external (bitmap) values into the correct internal values for the MegaPixel Display. **NXReadBitmap()** produces correct bitmaps from internal image representations.

Note: Premultiplication toward white is accomplished, in part, by using an internal gray scale that's the inverse of the PostScript gray scale. Internally, 0 is white and 1 is black.

At first glance, the difference between external and internal data representation may seem unnecessary. However, premultiplication toward black for external data representation is a device-independent standard. Converting from it to a variety of device-internal representations—including the representations required for color systems and the premultiplied-toward-white representations required for the MegaPixel Display—are easy and accurate. Adopting it as a standard means that you won't have to modify bitmap data to render it on different devices.

Compositing

In general, compositing combines two rectangular images of the same size, shape, and orientation, one a *source image* and the other a *destination image*, with the result replacing the destination image. Source and destination images may be located in the same window, or in different windows. The windows may be on-screen or off.

Compositing Operators

There are three compositing operators—**composite**, **compositorrect**, and **dissolve**. The most general of the three is **composite**. It and **compositorrect** are described below. **dissolve** is a special-purpose operator described later under “Dissolving.”

composite

The **composite** operator takes a list of eight operands; the first four specify the rectangle of the source image:

src_x src_y width height srcgstate dest_x dest_y op composite

The fifth operand, *srcgstate*, names a graphics state that specifies both the window device where the source rectangle is located and the coordinate system it's defined in.

The next two operands, *dest_x* and *dest_y*, locate the destination image in the current window and current coordinate system. Pixels in the destination image are paired one-to-one with pixels in the source image. The destination rectangle will be the same size and orientation as the source rectangle, regardless of the current coordinate system. It's positioned relative to (*dest_x*, *dest_y*) exactly as the source rectangle is to (*src_x*, *src_y*). This is illustrated in Figure 4-17.

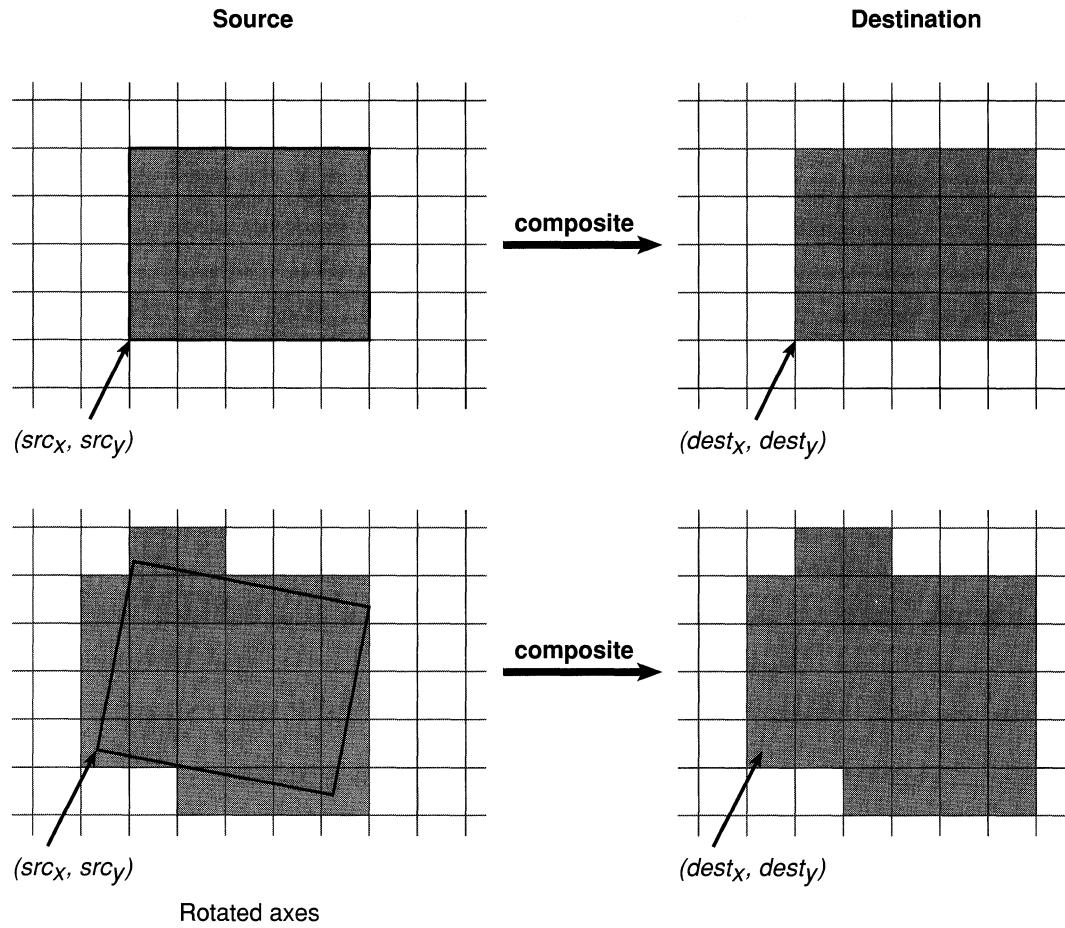


Figure 4-17. Compositing

The outline of the source rectangle may cross pixel boundaries due to fractional coordinates, scaling, or (as in the second example in Figure 4-17 above) rotated axes. The source image includes all the pixels that the rectangle encloses or enters. See “Imaging Conventions” later in this chapter for information on how outlines chose pixels.

The final operand to **composite** specifies the type of compositing operation. There are several to choose from:

Copy	Sover	Dover
Clear	Sin	Din
PlusD	Sout	Dout
PlusL	Satop	Datop
	Xor	

The **dpsNeXT.h** header file defines matching constants that can be used in Objective-C code:

NX_COPY	NX_SOVER	NX_DOVER
NX_CLEAR	NX_SIN	NX_DIN
NX_PLUSD	NX_SOUT	NX_DOUT
NX_PLUSL	NX_SATOP	NX_DATOP
	NX_XOR	

The operation called “Copy” is one of the most basic; it simply replaces the destination image with the source image. In the example below, it moves the source image ten units higher in the same window:

```
myGState setgstate      % make myGState the current graphics state
0 0 200 200             % the source rectangle
myGState                 % the source graphics state
0 10                     % location of the destination rectangle
Copy composite           % copy source to destination
```

In this example, the source and destination images overlap. This is typically the case when an image is scrolled.

compositerect

When the source of a compositing operation is a constant color, it’s convenient to use the **compositerect** operator instead of **composite**.

dest_x dest_y width height op compositerect

compositerect is like **composite**, except that there’s no real source image; its first four operands define the destination rectangle directly in the current window. **compositerect**’s effect on the destination is as if there were a source image filled as specified by the color and alpha parameters in the current graphics state. For example, you could erase a rectangular area to white by setting the current color to 1 and then calling **compositerect** to perform a Copy operation:

```
1 setalpha
1 setgray
0 0 100 100 Copy compositerect
```

This is exactly the same as:

```
1 setalpha
1 setgray
0 0 100 100 rectfill
```

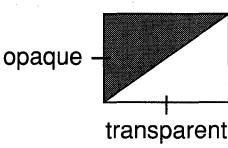
However, **compositerect** can be used for compositing operations other than Copy; **rectfill** and the other painting operators only render images in Copy mode.

Types of Compositing Operations

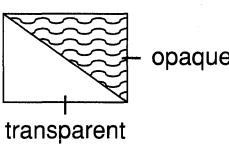
Most compositing operations are designed to make use of transparency in the source or destination image. But some, such as Copy illustrated above, are useful even when both images are completely opaque.

Figure 4-18 below illustrates all the compositing operations and summarizes each one in terms of what replaces the destination image. “Source image” or “destination image” in a result means both the color and coverage components of the pixels in the corresponding image. For simplicity, these summaries assume that the coverage values in the images are either 1 or 0 (opaque or fully transparent). The paragraphs following the illustration describe each operation in more detail.

Source



Destination before



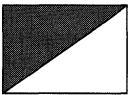
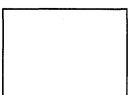
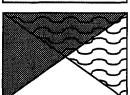
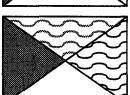
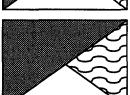
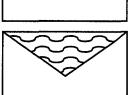
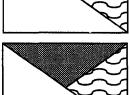
Operation	Destination after
Copy	 Source image.
Clear	 Transparent.
PlusD	 Sum of source and destination images, with color values approaching 0 as a limit.
PlusL	 Sum of source and destination images, with color values approaching 1 as a limit. (PlusL is not implemented for the MegaPixel Display.)
Sover	 Source image wherever source image is opaque, and destination image elsewhere.
Dover	 Destination image wherever destination image is opaque, and source image elsewhere.
Sin	 Source image wherever both images are opaque, and transparent elsewhere.
Din	 Destination image wherever both images are opaque, and transparent elsewhere.
Sout	 Source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Dout	 Destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.
Satop	 Source image wherever both images are opaque, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.
Datop	 Destination image wherever both images are opaque, source image wherever source image is opaque but destination image is transparent, and transparent elsewhere.
Xor	 Source image wherever source image is opaque but destination image is transparent, destination image wherever destination image is opaque but source image is transparent, and transparent elsewhere.

Figure 4-18. Compositing Operations

For a more complete discussion of compositing, including information on how composite images are calculated, see the paper “Compositing Digital Images” by Thomas Porter and Tom Duff in *Computer Graphics (SIGGRAPH '84 Conference Proceedings)* Volume 18, Number 3 (July 1984).

Copy

Copy is the simplest compositing operation and probably the one most used. It ignores the destination image, and replaces it with the source image; every pixel in the source rectangle is copied to the destination. Copy is used in scrolling to move an image from one location to another. It’s also used to bring images stored in off-screen windows to the screen. Copy is the only way images are moved on the NeXT computer.

Clear

Whereas Copy ignores the destination image, Clear ignores both the destination and the source. It turns the destination rectangle completely transparent. Clearing the rectangle has the same effect as painting it with pure transparent paint. This code

```
0 0 100 100  
Clear compositerect
```

is equivalent to

```
0 setalpha  
0 0 100 100 rectfill
```

except that Clear doesn’t change the alpha value in the current graphics state.

Clear is useful for creating a transparent surface to paint on. A transparent surface is like a clear sheet of plastic that you can selectively color with opaque paint. When the sheet is placed over another image, the other image will show through wherever paint hasn’t been applied.

Before it’s painted on, a transparent surface lets the background color—white on the MegaPixel Display—show through everywhere. However, you should never use Clear simply to erase to the background. A transparent surface isn’t the same as an opaque white surface. If the destination window doesn’t already have memory allocated to store the coverage component of its pixels, Clear will allocate it. It may therefore double the amount of memory for the window.

PlusD and PlusL

PlusD (for “darker”) and PlusL (for “lighter”) are two versions of the same basic operation. They both add the source and destination images together. Where partially transparent areas in the source and destination overlap, the composite becomes more opaque. It’s like looking through two panes of tinted glass rather than one; together they’re more opaque than either alone. Where either of the original images is completely opaque, the composite is also opaque; where either image is completely transparent, the composite has the coverage of the other image.

PlusD and PlusL differ only in how they add source and destination color values. PlusD adds colors so that they become darker; PlusL adds them so that the result is lighter.

PlusD compositing is the more natural of the two for devices like the MegaPixel Display where the white background of the PostScript imaging model is assumed; it adds color values to be less like the background. Where the gray of one image overlaps the gray of the other, a darker shade of gray results. Black plus any other color yields black; white plus another color yields the other color.

PlusL compositing is not currently implemented for the MegaPixel Display. It would be most naturally used where a black background is assumed, for it adds color values to become lighter, less like the background. Where the gray of one image overlaps the gray of the other, a lighter shade of gray results. Black plus any other color yields the other color; white plus another color yields white.

Transparency Operations

For Copy, Clear, PlusD, and PlusL, neither the source nor the destination image need be transparent (though they can be). In contrast, the other compositing operations are most interesting and useful when one (or both) of the images is partially transparent. The transparency of one image helps determine what you’ll see of the other.

With one exception, the operations that make use of transparency come in pairs and have similar names. The first letter of the name is an “S” or a “D,” standing for “source” or “destination.” The remaining letters of the name describe the type of operation. They’re summarized in the list below:

Sover and Dover One image is placed over the other. Transparency in the image on top lets the image underneath show through.

Aside from Copy, Sover and Dover are perhaps most common and useful compositing operations. They permit a foreground image to be composited over a background image, or let areas of partial opacity in the image on top add shading to the image underneath. The arrow cursor you see on the screen is placed there using Sover compositing. The source rectangle has transparent pixels surrounding the opaque arrow.

Sin and Din	One image is displayed wherever, and to the extent that, the other image is opaque. In a sense, the transparency of the second image eliminates portions of the first.
	Sin and Din can be used to clip a picture to a particular shape, say an oval. The oval is drawn with opaque paint on a clear (transparent) surface, and the picture is Sin'ed over it. Or, if you've drawn a portrait of someone on a clear surface, you can turn it into a silhouette by using compositerect to Sin a black rectangle over it. All the opaque pixels in the portrait will turn black.
Sout and Dout	One image is displayed wherever, and to the extent that, the other image is transparent. The opacity of the second image eliminates portions of the first.
Satop and Datop	One image is placed on top of the other, with the composite adopting the transparency of the image underneath. The image on top shows through only where (and to the extent that) the image underneath is opaque. The image underneath shows through only where (and to the extent that) it's opaque and the image on top is transparent.
Xor	Each image is visible only where (and to the extent that) it's opaque and the other image is transparent.

Dissolving

The **dissolve** operator blends two images together. Typically, it's called over and over again in a loop so that one image (the source) can appear to slowly replace the other (the destination).

dissolve takes almost the same set of operands as **composite**, but since it uses a compositing operation that's particular to its purpose (one similar to Plus), the operation isn't stated as an operand. Instead, you specify a fraction, *delta*, that determines how much each image contributes to the composite:

```
src_x src_y width height srcgstate dest_x dest_y delta dissolve
```

When *delta* is 0, only the destination image is in the composite; when it's 1, only the source image is present. In the example below, 40% of the composite comes from the source image; the remaining 60% belongs to the original destination image.

```
0 0 200 200 myGState 50 50 0.4 dissolve
```

To gradually replace one image with another, **dissolve** should be called in a loop with *delta* increasing slowly from 0 to 1 with each call. Since it must combine the original destination and source images each time, the altered destination image must be replaced with the

original before **dissolve** is called. This is done by storing the destination image in an off-screen window and copying it to the destination rectangle each time through the loop. Dissolving should be done in a buffered window so that only the result produced by the **dissolve** operator is flushed to the screen.

In the example below, the original destination image is stored in an off-screen window identified by the **offGS** graphics state object; the source image is in a window identified by the **srcGS** graphics state object.

```
0 1 64 {                                % begin loop, stepping from 0 to 64

/delta exch 64 div def    % redefine delta each time as the current step
                           % number of steps divided by the total

0 0 200 200 offGS
0 0 Copy composite      % copy in the original destination

0 0 200 200 srcGS
0 0 delta dissolve       % dissolve in the source image

flushgraphics             % flush the result to the screen

} for                      % end the loop
```

Highlighting

On the NeXT MegaPixel Display, highlighting is usually accomplished by changing white pixels to light gray and light gray pixels to white:

- Buttons and menu commands have a light gray background that turns to white when they're highlighted.
- Selectable text is displayed against a white background, which becomes light gray to mark the selection.

Since the background color to a display is typically light gray or white, this type of highlighting fits a variety of contexts and doesn't radically change what the user sees.

To make highlighting easy, there's an additional compositing operation, called "Highlight," that's used only with the **compositerect** operator:

```
10 10 50 50 Highlight compositerect
```

On the MegaPixel Display, Highlight turns every white pixel in the destination rectangle to light gray and every light gray pixel to white, regardless of the pixel's coverage value. Repeating the same operation reverses the effect. (Highlight may act differently on other devices. For example, on displays that assign just one bit per pixel, it would invert every pixel.)

The **NXHighlightRect()** function is the most direct way to highlight in Objective-C code. It performs Highlight compositing on the area specified by an NXRect structure. The code in the example below is equivalent to the PostScript code shown above:

```
NXRect rect = {{10.0, 10.0}, {50.0, 50.0}};  
NXHighlightRect(&rect);
```

Note: The Highlight operation doesn't change the value of a pixel's coverage component. To ensure that the pixel's color and coverage combination remains valid, Highlight operations should be temporary and should be reversed before any further compositing.

Instance Drawing

A screen-oriented user interface often does highly interactive drawing as a means of providing feedback to the user, typically in response to the user dragging with the mouse. Examples of this include stretching out a rectangle, moving an object around on the screen, and highlighting objects as they're selected. In general, something is drawn over and over again, changing its location, size, or orientation each time, with previous occurrences of the image disappearing as new occurrences are drawn. This kind of interactive, temporary drawing is facilitated by a NeXT extension to the Display PostScript system known as *instance drawing*.

Instance drawing is temporary drawing done within an on-screen window. Rendered images appear directly on-screen and aren't saved in the window's buffer. They can be easily removed and replaced with the original image, usually in preparation for the next instance drawing.

The **setinstance** operator turns instance drawing mode on and off. While the mode is on, all drawing in the current window is treated as instance drawing rather than as part of the window's permanent display. The mode setting is stored as a parameter in the current graphics state. All images produced in instance drawing mode are removed by the **newinstance** operator (so named because you call it when you're about to replace a previous instance with a new one). **newinstance** restores the original image.

In addition, there's a **hideinstance** operator for restoring the original image within a specified rectangle. For example, if instance drawing is used to mark selected objects, **hideinstance** can be invoked to unmark objects the user has deselected by dragging back over them.

If you do instance drawing in a retained or buffered window and part of the drawing is obscured by another window, you'll get a window-exposed subevent (of the kit-defined event) if the obscured part of the drawing is exposed. In practice, it should rarely happen that your instance drawing is obscured by another window; usually you'll do instance drawing in the frontmost window, since that's the window the user is actively working in. (Events are discussed in the next chapter.)

The following example illustrates the three instance drawing operators—**setinstance**, **hideinstance**, and **newinstance**. You can run the code that's presented here in a PostScript previewer such as **pft** to see the demonstration on-screen.

Normally you'd do instance drawing in response to events and take some of the drawing coordinates from the current position of the cursor. But this demonstration uses coordinates that are randomly determined. In a self-contained loop, it draws a black rectangle in a narrow window along the left edge of the screen and bounces one side of the rectangle up and down. The bouncing effect is produced by progressively adding small slices to the rectangle to make it grow larger, and by subtracting small slices to make it grow smaller. Slices are added in instance drawing mode and hidden using the **hideinstance** operator.

First, the window is created:

```
/w 100 def           % Define w for "width"  
/h 832 def           % Define h for "height"  
0 0 w h Buffered window  
windowdeviceround  
Above 0 currentwindow  
orderwindow           % Create a buffered window  
                      % Make it the current device  
                      % Place the window on-screen
```

To show that **hideinstance** and **newinstance** reinstate the original image, there has to be something showing in the window before instance drawing begins. This code supplies a simple design:

```
0 0 0 h w h w 0           % Put window corners on the stack  
3 {8 copy 8 2 roll} repeat  
4 {moveto curveto} repeat  
stroke  
flushgraphics             % Set up to start from each corner  
                          % Draw the curve from each corner  
                          % Stroke the path  
                          % Flush it to the screen
```

The code that follows draws the bouncing rectangle. It has two loops: The outer one switches the end of the window where the rectangle is drawn. The inner loop expands and contracts the rectangle from that end a random number of times. Each time through the inner loop, a new y coordinate (**ynew**) is selected at random. Depending on the location of **ynew**, the rectangle is either stretched—by having **rectfill** repeatedly fill one-pixel-high cross sections of the window—or constricted—by having **hideinstance** remove one-pixel-high cross sections.

After a few times, this whole process is repeated at the other end of the window. To start fresh from the other end, the **newinstance** operator is used to remove the entire rectangle.

```

true setinstance
50 {
    /yold 0 def
    rand 20 mod {
        /ynew rand
        h mod def
        ynew yold gt {
            yold 1 ynew
            {0 exch w 1
            rectfill} for
        }{
            % Otherwise, hide slices of the rectangle
            yold -1 ynew
            {0 exch w 1
            hideinstance} for
        } ifelse
        /yold ynew def
    } repeat
    0 h translate
    1 -1 scale
    newinstance
} repeat
false setinstance
% Turn on instance drawing mode
% Repeat the outer loop 50 times
% To start, define yold to be 0
% Repeat inner loop 0-19 times
% Define ynew to be a random number less
% than the window height
% If ynew is greater than yold, add
% slices to the rectangle
% Otherwise, hide slices of the rectangle
% Redefine yold to be ynew
% Repeat the inner loop
% Move the coordinate origin
% Flip the polarity of the y-axis
% Remove all instance drawing
% Repeat the outer loop
% Turn off instance drawing mode

```

Sending PostScript Code to the Window Server

Your application must send PostScript code to the NeXT Window Server to draw on the screen. Communication with the Window Server is made possible through a bidirectional connection between the application process and the Window Server process. This section discusses how applications send drawing information over this connection to the Server.

Programs based on the Application Kit establish this connection automatically at startup. If your program doesn't use the Application Kit, you must establish a connection to the Window Server through direct calls to functions defined in the library **libNeXT.a**. This library includes the basic Display PostScript client library along with extensions that support the NeXT window system.

The primary documentation for the client library and for the Display PostScript System in general is contained in a series of manuals by Adobe Systems. See "Suggested Reading" in the *NeXT Technical Summaries* manual for information about these manuals. For information about those functions that are specific to the NeXT implementation of the Display PostScript System, see *NeXTstep Reference, Volume 2* and the comments in the NeXT header file **dpsclient/dpsNeXT.h**.

Once a connection is established, information passes across it in both directions until the connection is closed when the user quits the application. For transmission efficiency, the information sent to the Window Server consists primarily of binary-encoded PostScript code. For further efficiency, the information passed in either direction across the

connection is accumulated into buffers before being sent across the connection. In most cases, the buffers are flushed automatically without the explicit intervention of the application.

Most of the drawing information your application sends to the Window Server (for example, instructions for drawing standard user-interface objects like buttons and sliders) is created and sent for you by the Application Kit objects that make up your application. You only need to write and send drawing instructions for those features that are unique to your application. There are several ways to accomplish this: using the **pswrap** program, using C functions that correspond to individual PostScript operators, or writing directly to the connection. The next three sections discuss these techniques.

Using **pswrap**

For most programmers, **pswrap** provides the best way of sending PostScript code to the Window Server. **pswrap** (described in detail in Adobe Systems' *pswrap Reference Manual*) is a program that creates a C function to correspond to a sequence of PostScript code. When your application is run, a call to the C function sends a binary-encoded version of the PostScript code to the Window Server.

Each function definition begins with **defineps** and ends with **endps**. The definition includes the C function name, any the function requires, and a listing of the PostScript code the function represents.

For example, you could define a function that draws a square on the screen, as illustrated below:

```
defineps drawSquare(float x)
    newpath
    100.0 100.0 moveto
    0.0 50.0 rlineto
    50.0 0.0 rlineto
    0.0 -50.0 rlineto
    closepath
    x setgray
    stroke
endps
```

The first six lines of PostScript code in this example outline a square area on the screen. The **setgray** operator sets the shade of gray equal to the value of **x**, the input argument to this function. Finally, **stroke** paints the outline with the selected shade of gray.

Elsewhere in your C source code, you can call this **drawSquare** function in the same manner as you would any C function:

```
drawSquare(0.5);
```

pswrap lets you combine PostScript and Objective-C (or C) code in a natural way. **pswrap** functions can take input arguments (as above) and can, through output arguments, return values sent across the connection from the PostScript interpreter. A **pswrap** function can be defined in a separate file or can be included in the same file as the Objective-C source code. If Objective-C and PostScript code are combined in a single file, **pswrap** passes the Objective-C code through unchanged to an output file and translates only the embedded PostScript code that's marked by the **defineps** and **endps** delimiters. After preprocessing with **pswrap**, the output file can be compiled in the normal way.

The benefits of using **pswrap** to create a function corresponding to several lines of PostScript code are most clearly seen in comparison to the alternatives: the single-operator functions and writing directly to the connection.

Using Single-Operator Functions

The easiest way to send one or two lines of PostScript code to the Window Server is by using a set of C functions that have a one-to-one correspondence with the PostScript operators. These functions were created by applying **pswrap** to PostScript code containing each of these operators; thus, they can be considered a fundamental kind of **pswrap**-generated function.

Each of these “single-operator” functions has the same name as the corresponding PostScript operator, but begins with the prefix “PS” or “DPS”. The functions that have the “PS” prefix act on the current PostScript context; you must specify the PostScript context for the “DPS” functions.

Note: Standard Display PostScript single-operator functions are listed in Adobe Systems’ *Client Library Reference Manual*. Those single-operator functions that are specific to the NeXT computer environment are described in *NeXTstep Reference, Volume 2*. The C functions chapter in the *Technical Summaries* manual summarizes all single-operator functions.

An example of a single-operator function is **PSsetgray()**, the C function that corresponds to the PostScript **setgray** operator.

The arguments to a single-operator function are the operands required by the corresponding PostScript operator, as seen in this rewrite of the above example:

```
PSnewpath();
PSmoveto(100.0, 100.0);
PSrlineto(0.0, 50.0);
PSrlineto(50.0, 0.0);
PSrlineto(0.0, -50.0);
PSclosepath();
PSsetgray(0.5);
PSstroke();
```

In some cases, if a PostScript operator leaves a value on the stack, the corresponding single-operator function also leaves a value on the stack rather than returning it to the application. This is done for efficiency, since typically the value is needed as input for the next operator to be executed.

In general, using **pswrap** to create a customized function corresponding to multiple lines of PostScript code is preferable to using multiple single-operator functions. Customized functions reduce the interpretation overhead incurred by multiple single-operator function calls. In addition, **pswrap** lets you program directly in PostScript code, making program listings more natural and concise.

Connection Buffering

For efficiency, data passed from the application to the Window Server is first accumulated in a buffer. Data returned from the Server is similarly buffered. To ensure that all data in the application's buffer is sent across the connection, the buffer must be flushed. As long as your application makes use of Application Kit facilities, you will rarely need to flush buffers explicitly.

The buffer on the application side is flushed automatically at the following times:

- Whenever the application tries to get events
- After a timed entry executes

This buffer can also be flushed explicitly with a call to **DPSFlush()**. Simple programs that just draw on the screen but don't get data from the Window Server have to call **DPSFlush()** to ensure that all PostScript code generated is flushed to the Server. Applications might also call **DPSFlush()** in these situations:

- Before doing especially time-consuming processing in response to an event, to give the user more timely feedback
- In GDB (the GNU debugger), to ensure that all PostScript code generated so far has been sent to the Window Server

The buffer on the Window Server side is flushed automatically at these times:

- Whenever the Server has an event or error to send the application
- Whenever the **flush** operator is executed

Connection buffering and the fact that your application and the Window Server are separate processes executing asynchronously provide another reason for using **pswrap** functions to send drawing information to the Window Server. Code destined for the Window Server is not sent until the application-side buffer is flushed. Once the code arrives in the Server, it may have to wait to be executed until the Server has finished with another application's input. Finally, tokens are returned from the Server as they become available; token output is not synchronized with PostScript code input.

Fortunately, **pswrap** insulates you from most of the consequences of this asynchrony. As described above, a **pswrap** function flushes the application-side buffer for you. It also flushes the Window Server side buffer when a token is ready to be returned to the application. The returned token is checked for an internal synchronization code to ensure that the token is the correct one to return to a particular **pswrap**-generated function. In fact, the only time this asynchrony of execution may become evident is when error messages are received. An error message received at one moment may refer to a section of code sent to the Server somewhat earlier. However, error messages usually contain enough contextual information to allow you to determine the problem's location relatively easily.

Imaging Conventions

In the PostScript language, all visible forms have an outline delineating the area to be colored. This is true both of filled figures and of simple lines.

There are two steps to drawing a filled figure:

1. Construct a path around the area that's to be colored.
2. Fill the area outlined by the path.

Drawing a line—whether it's open (like an arc) or closed (like a full circle)—also takes two steps:

1. Construct the path where you want the line located.
2. Stroke the path to color the line.

Stroking fills an outline that's constructed along either side of the path using the line width, line cap, and line join parameters of the current graphics state. One-half the line width falls on one side of the path and one-half on the other. This outlines the area covered by the line, with the path running through the center of the outline. The line cap parameter determines the shape of the outline at the ends of an open path; line join determines its shape where separate segments of the path meet. When the path is stroked, the outline is filled. The path itself has a width of 0.

Figure 4-19 illustrates the path and outline of a short curved line with a width set to 2 and butt-end line caps.

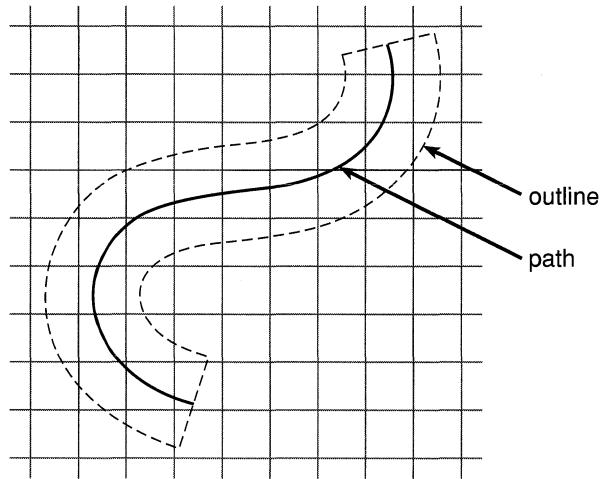


Figure 4-19. Path and Outline

It's possible that a line drawn in this way will totally cover only a few pixels. A one-pixel-wide line drawn along a path from (4.0, 10.0) to (700.0, 183.0), for example, will touch many pixels, but cover none of them entirely.

Because figures and paths can cut across pixels in this way, it's reasonable to ask just which pixels an outline will affect. For the most part, the answer doesn't matter: Pixels are so small that you ordinarily don't need to be concerned with which ones are turned on when an outline is filled.

For the cases when you're doing detailed drawing and are concerned with this question, the remainder of this section discusses the way imaging is carried out on the NeXT MegaPixel Display and 400 dpi Laser Printer.

The General Rule

An outlined figure selects all the pixels that it overlaps. This includes all the pixels its outline completely surrounds and any pixel the outline enters, even if most of the pixel actually lies outside the area the outline encloses. (An outline enters a pixel if it crosses from one side to another, or if it simply cuts into the pixel without actually passing through.)

This general rule chooses the pixels that can be colored—either uniformly or in a halftone pattern—when an outline is painted. The examples below illustrate this principle. Each assumes that the drawing is done with solid, opaque paint; the pixels that are colored in the illustrations are the ones this rule selects.

Consider the outline of a white circle, like the one illustrated in Figure 4-20. The path defining the circumference of the circle completely encloses some pixels, but encloses only parts of others. Every pixel the circumference passes through is turned white, even if most of the pixel lies outside the circle.

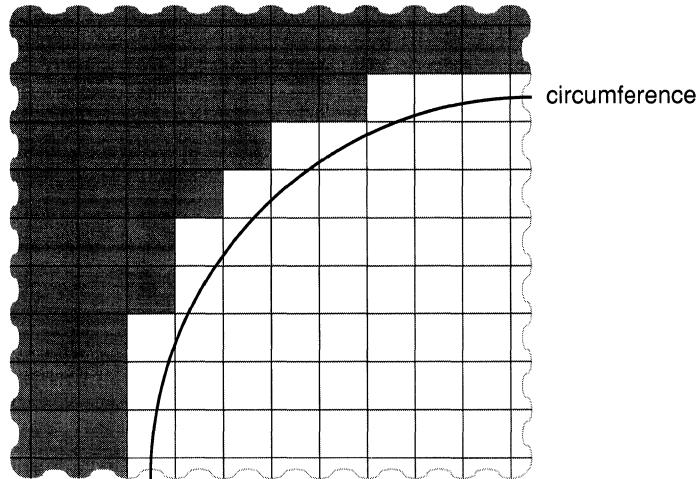


Figure 4-20. Section of a White Circle

If an outline lies on a pixel boundary but doesn't enclose or enter the pixel, the pixel isn't treated as if it were inside the outline. The pixel and the outlined figure don't overlap.

The 3-by-5 rectangle illustrated in Figure 4-21 has an outline that doesn't enter any pixels; it lies entirely on pixel boundaries. When the rectangle is filled with black paint, only the 15 pixels entirely enclosed within the outline turn black.

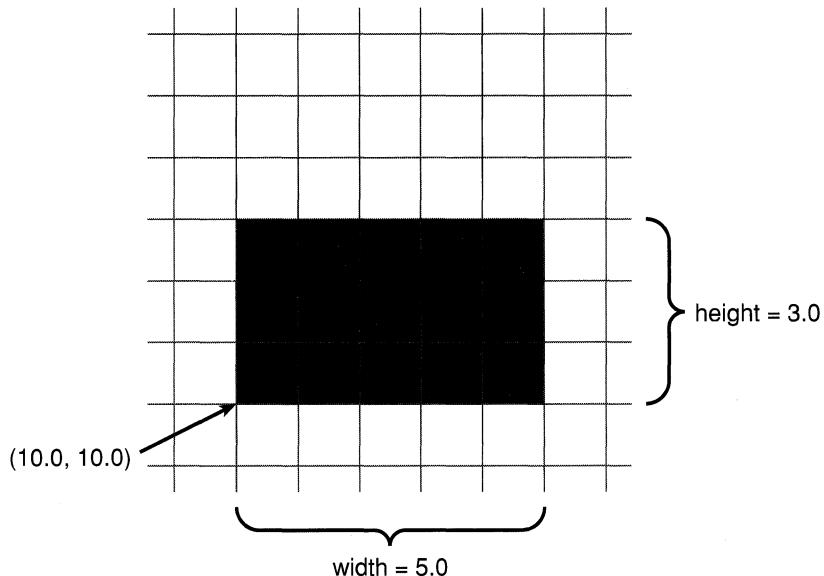


Figure 4-21. Filled Rectangle

Figure 4-22 and Figure 4-23 illustrate two horizontal lines, each with its line width set to 1. The path of the line in Figure 4-22 was drawn down the center of a row of horizontal pixels—say, from (100.0, 4.5) to (800.0, 4.5). Its outline completely covers each pixel on its path, but it doesn't enter any in the adjacent rows. Therefore, only a single row of pixels is colored to show the line.

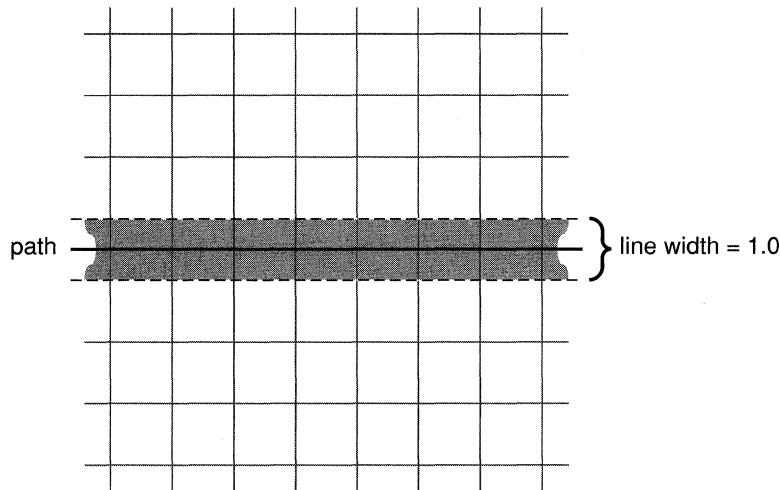


Figure 4-22. Horizontal Path at Pixel Midpoint

In Figure 4-23, the path of the line was drawn slightly lower—say, from (100.0, 4.3) to (800.0, 4.3). The line partially overlaps two rows of pixels, the row the path passes through and the row beneath. Both rows are colored to display the line.

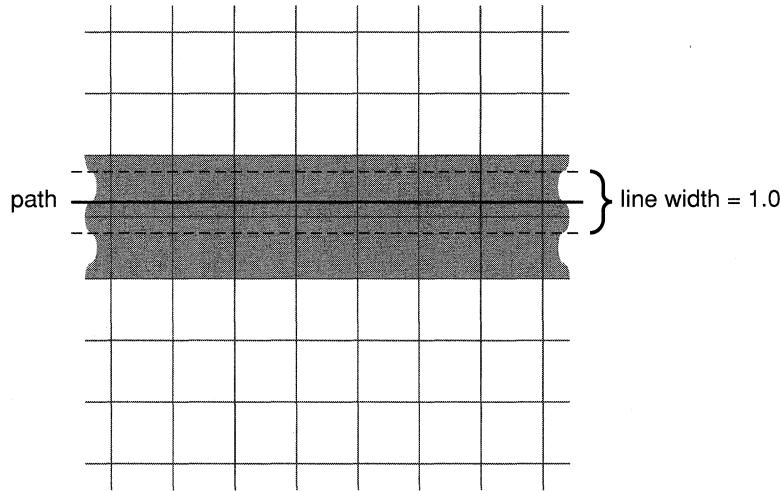


Figure 4-23. Horizontal Path below Pixel Midpoint

As the two illustrations above show, an outline one-pixel wide will be displayed as a one-pixel line only if it falls midway between pixel boundaries. Otherwise, it will be twice as thick.

This variation in how outlines of the same width are displayed is generally not desirable. It's especially noticeable on-screen and especially when the outlines are thin.

To eliminate the variation, the Display PostScript system has added a parameter to the graphics state which, when true, adjusts the thickness of stroked lines so that they always approximate the width of the outline. With stroke adjustment, the line shown in Figure 4-23 above would turn on a single row of pixels, no matter where it's located. This is illustrated in Figure 4-24.

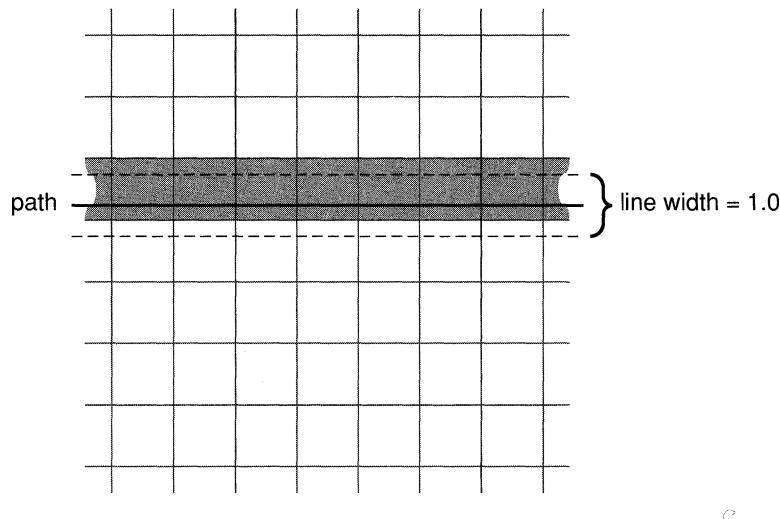


Figure 4-24. Stroke Adjustment for a Path below Pixel Midpoint

Stroke adjustment is true by default on the NeXT computer.

The general rule, as stated and illustrated above (and as modified by stroke adjustment), is all that's required to select pixels if the outline encloses any area at all. Where an outline collapses to a single point or to a line with a width of 0, however, additional rules are needed. The additions are required so that:

- A zero-width line can be shown using the fewest number of pixels possible.
- All figures—even points and zero-width lines drawn along pixel boundaries—can be visible.

The following two sections discuss these special rules.

Outlines with No Area

When, to draw a very thin line, you set the PostScript line width variable to 0, the resulting outline is identical to the path; it has no width at all. A zero-width outline can also result when a path that defines a filled figure doubles back on itself. Occasionally, an outline will collapse to a single point.

Points

A single point is shown by turning on a single pixel. If the point lies in the middle of a pixel, that pixel is the one turned on. If the point lies between pixels, just one of the pixels is chosen. The choice depends on the device. Figure 4-25 below illustrates how the choice is made on the NeXT MegaPixel Display and 400 dpi Laser Printer.

- If the point lies on the vertical boundary between pixels, the pixel on the right is the one chosen.
- If the point lies on the horizontal boundary between pixels, the one below is chosen.
- If the point lies on the corner of four pixels, the one on the lower right is chosen.

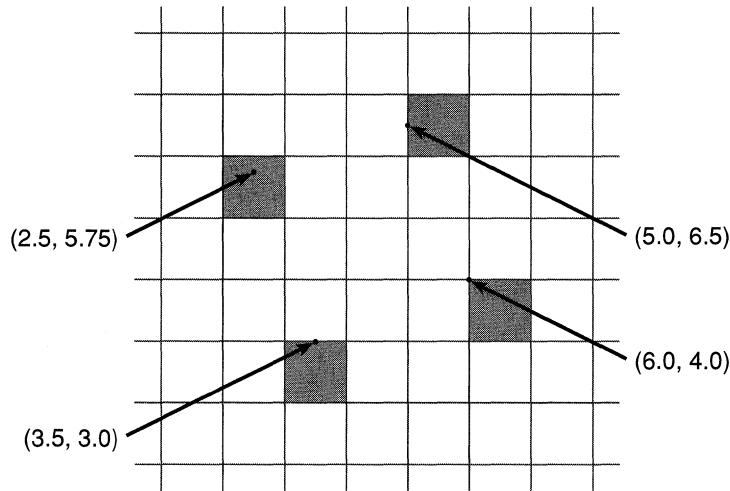


Figure 4-25. Pixels that Display Points

Zero-Width Lines

Zero-width lines are rendered by turning on the fewest possible number of pixels to show a connected line. For a straight line, this is accomplished by comparing the base coordinates of its two endpoints:

- If the difference between the two x coordinates is greater than the difference between the two y coordinates (the line is more horizontal than vertical), exactly one pixel will be turned on in each vertical column between the two endpoints.
- If the difference between the two y coordinates is greater than the difference between the two x coordinates (the line is more vertical than horizontal), exactly one pixel will be turned on in each horizontal row.
- If the differences between the y coordinates and the x coordinates are the same (the line is drawn at a 45° angle), exactly one pixel will be turned on in each column and in each row.

Figure 4-26 illustrates how these conventions apply to a variety of straight zero-width lines.

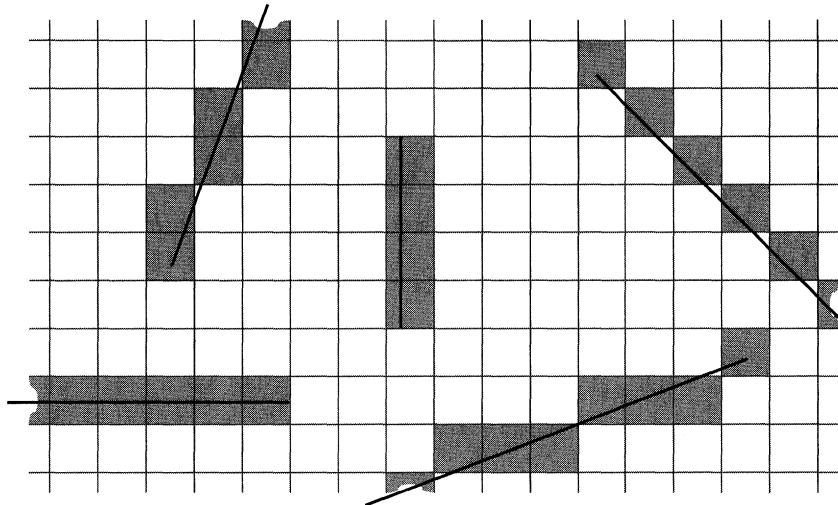


Figure 4-26. Zero-Width Lines

To choose which pixel to turn on in a particular column or row, the conventions look only at the section of the line falling within that column or row. The pixel that includes the section's midpoint is the one that's chosen. Figure 4-27 shows how a pixel is chosen when a zero-width path passes through two pixels in the same column. In the illustration, the pixel on the top left is chosen over the pixel below it, because it contains the midpoint of the path segment falling within the left column.

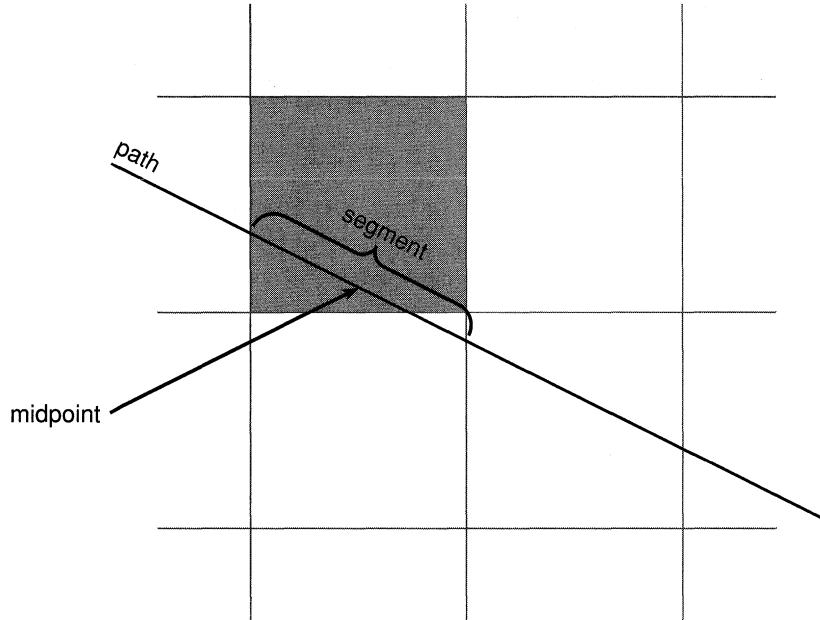


Figure 4-27. Choosing a Pixel

When the midpoint falls on the boundary between pixels, only one of the pixels is chosen. The choice is made for midpoints in the same way as it's made for other points (see "Points" and Figure 4-26).

- Within a row, the pixel on the right is chosen.
- Within a column, the lower of the two pixels is chosen.

Where a horizontal or vertical zero-width line lies entirely between pixels, every midpoint is on a pixel boundary. This is illustrated by the rectangular path in Figure 4-28. Here the line width was set to 0 and the path was stroked rather than filled. The pixels that are colored are the ones that would be chosen on the NeXT screen and laser printer.

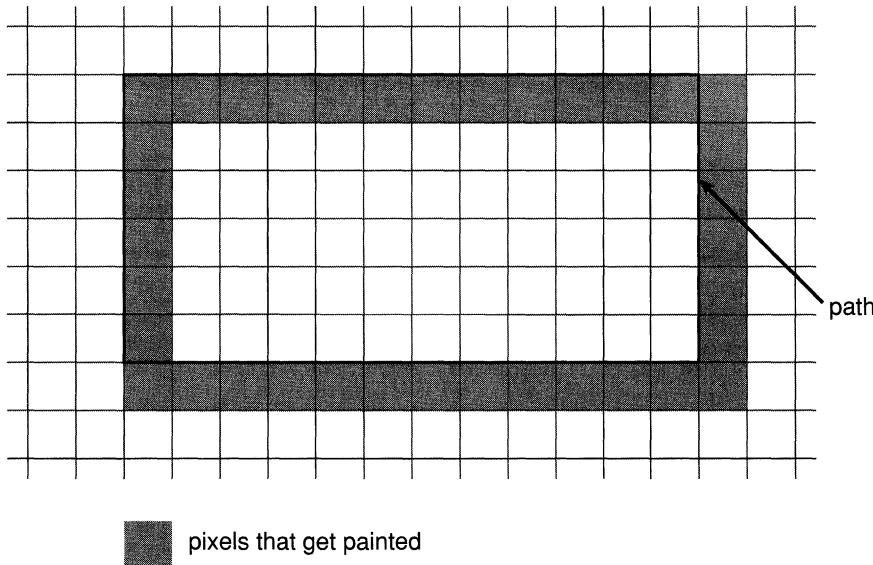


Figure 4-28. Zero-Width Line on a Rectangular Path

All the conventions discussed above for zero-width lines are used even if the outline isn't straight. An automatic PostScript flattening process turns a curved path into a series of very short, straight-line segments that approximate the curve. The conventions apply to these segments. (The segments are actually chords joining selected points along the curved path.)

Half-Open Shapes

This section explains the principle that lies behind the general rule and some of the other imaging conventions that were illustrated above. It's presented for readers who would like to know the basis for the conventions.

The general rule can be restated as follows:

If a pixel and an outlined figure have one or more points in common, the pixel is part of the figure.

It's fairly easy to tell that a pixel and a figure have a point in common when the point lies within the pixel and is surrounded by the outline of the figure. It's more problematic when the point is on a pixel boundary or on the outline itself.

To solve the problem, all shapes, whether pixels or outlined figures, are considered "half-open": Only about half the points on the edge of the shape are included in it. When two figures are tiled together, each point along their common border will belong to just one of the figures. Similarly, the points on a border between two pixels belong to just one pixel; each point on the screen is assigned to one and only one pixel.

Since PostScript flattening turns every curve into a series of short, straight segments, the rules that determine which points on the edge of a shape belong to the shape, and which do not, need to be stated only for straight edges. All shapes, whether outlines or pixels, are covered by the same rules:

- If the edge of a shape faces exactly upward or at all leftward, the points on the edge belong to the shape. If the edge faces exactly downward or at all rightward, the points on the edge don't belong to the shape.
- A point at the corner between two straight edges belongs to the shape if both of the edges do, or if just one edge does and the corner is concave. If neither edge belongs to the shape, the corner point doesn't either.

Note: The directions stated in these rules are for the NeXT MegaPixel Display and 400 dpi Laser Printer. They may be different on other devices.

By these rules, each pixel includes its upper and left borders and the corner point that joins them. The upper left is the only corner that belongs to the pixel.

The outline of a straight zero-width line collapses to the path of the line, but is still treated as having an edge on each of its sides. The edges face in opposite directions. Following the rules stated above, the endpoints of the line—the corners between the two edges—don't belong to the line.

Clipping

A PostScript clipping path outlines the area where painting operators (such as **show**, **fill**, and **stroke**) can render an image. Like any other outline, it selects pixels according to the general rule and other conventions discussed above. Only pixels selected by both the clipping path and an outlined figure can actually be used to display the figure.

Figure 4-29 and Figure 4-30 show how clipping paths and image outlines interact. Both figures illustrate a rectangular path like the one shown in Figure 4-28. This time, however, the width of the line is set to 1 and path is made to double as the clipping path.

In Figure 4-29, the path is located exactly as it was in Figure 4-28: entirely on pixel boundaries. The outline of the one-pixel-wide line therefore covers half of every pixel on either side of its path. However, none of the pixels around the outside of the clipping path are within the area where drawing can occur, since the clipping path doesn't enter or enclose them. Only a single strand of pixels inside the clipping path is colored to display the line.

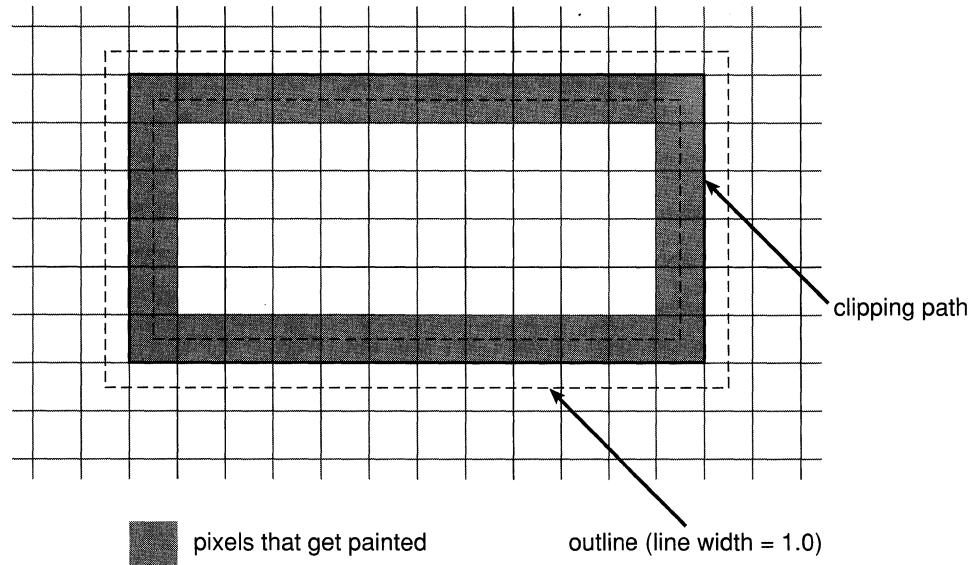


Figure 4-29. Clipping Path between Pixels

In Figure 4-30, the rectangle has been shifted slightly to the left. Each of the vertical sections of the clipping path now passes through a column of pixels. On the left, both columns of pixels the outline enters are used to display it. One column has the clipping path running through it, so it isn't clipped from the drawing area. The other column is entirely surrounded by the clipping path, so it also lies within the drawing area. (On the right, only one of the columns lies within the drawing area.)

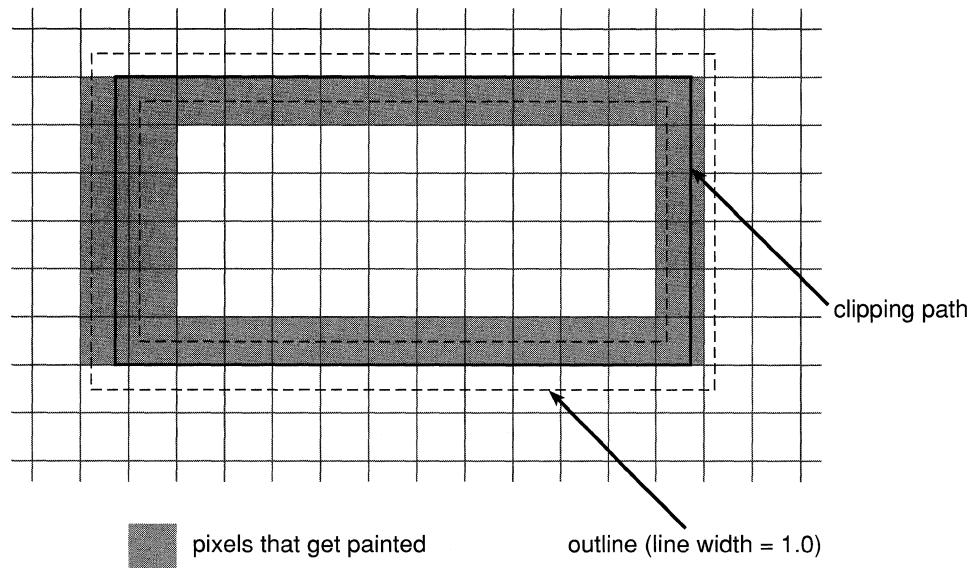


Figure 4-30. Clipping Path Crossing Pixels

Finally, it might be helpful to return to the zero-width line illustrated in Figure 4-28, one drawn along a rectangular path lying entirely between pixels. What happens if the path is made into a clipping path? Since the path and zero-width line don't enter any pixels, only the pixels that the clipping path entirely surrounds can be used to display the line. As shown in Figure 4-31, just two sides of the rectangle will be visible.

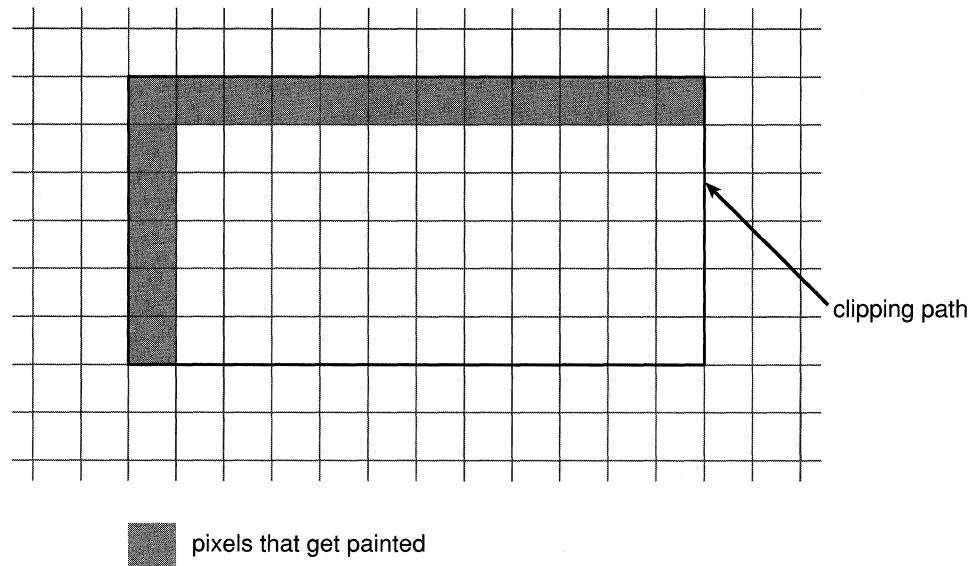


Figure 4-31. Clipped Zero-Width Rectangular Line

Chapter 5

Events

5-4 Event Basics

5-4	Types of Events
5-5	Keyboard Events
5-5	Mouse Events
5-6	Timer Events
5-7	Cursor-Update Events
5-7	Kit-Defined Events
5-8	System-Defined Events
5-8	Application-Defined Events

5-9 The Event Record

5-10	Event Types
5-11	Event Flags
5-13	Type-Specific Event Data
5-14	Mouse-Down and Mouse-Up Events
5-15	Key-Down and Key-Up Events
5-15	Mouse-Entered and Mouse-Exited Events
5-16	Kit-Defined Events
5-17	System-Defined Events
5-17	Application-Defined Events

5-17 Keyboard Information

5-18 Event Masks

5-20 The Event Queue

5-21 Event-Related Services

5-22	Executing Timed Entries
5-23	Interval Variability
5-24	Checking Mach Ports
5-24	Checking File Descriptors
5-25	Scheduling

Chapter 5

Events

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

**/NextLibrary/Documentation/NextDev/ReleaseNotes/WindowServer.rtf
/NextLibrary/Documentation/NextDev/ReleaseNotes/AppKit.rtf**

Applications on the NeXT computer respond to four types of input:

- Events. The direct or indirect reports of a user's actions—*key-down* and *mouse-up* events, for example.
- Timed entries. Functions that are executed at a given frequency.
- Data received at a Mach port.
- Data waiting at a file descriptor.

Of these four, events are by far the most prevalent and important. In fact, the heart of an application is its *event loop*, the routine that repeatedly checks for the arrival of new events. It's only when the application is running the event loop that it can receive input from the three other sources. Because these four types of input are so closely interrelated, they're discussed together in this chapter.

The chapter begins with an examination of events: their types, components, and how they're distributed to an application. It then discusses the other types of input and how an application coordinates its response to them.

Since most of an application's event processing is done through the Application Kit objects it includes, the close look at events that this chapter provides may at times contain more detail than you'll actually need to get started writing an application. The next two chapters, "Program Structure" and "Program Dynamics," introduce the Application Kit and its event-handling system. You can either turn directly to those chapters and refer back to this one for background information as needed, or you can read through this chapter first.

Event Basics

Applications on the NeXT computer are driven by the user's actions on the keyboard and mouse. The Window Server treats each user action as an *event*, which it associates with a window and reports to the application that created the window. For most events, the Window Server sends the event to the application only if the window's *event mask* permits it. The mask has a different bit set for each type of event the application is interested in.

Pertinent information about each event—such as which character was typed and where the mouse was located—is collected in an *event record*. When an event is reported to an application, its record is stored in a C structure and made available to the application through the Application Kit.

Since an application and the Window Server are separate processes that execute asynchronously, a system is needed to coordinate their communication. For example, an application may be involved in a long computation and momentarily unable to process the event records that the Window Server sends. So that no information is lost, as event records are received in the application, they're temporarily placed in storage called the *event queue*. When the application is ready to process an event, it takes an event record from the queue. Only when the application checks the queue can it also check for timed entries or data at a Mach port or in a file.

Most events follow the same path: from the Window Server to the application's event queue, and from there, to the objects and functions of the application. However, the Application Kit can create an event record and insert it into the event queue for distribution or send it directly to its destination. In some cases, it's more efficient or convenient for the Application Kit to create an event record itself. The Kit can then either insert the record in the event queue for distribution or send it directly to its destination. If an event follows one of these alternate pathways, it's pointed out in the discussion that follows.

Types of Events

The different types of events an application receives can be grouped in these seven categories:

- Keyboard events
- Mouse events
- Timer events
- Cursor-update events
- Kit-defined events
- System-defined events
- Application-defined events

As described in the following sections, some of these categories comprise *discrete events*. For example, mouse-down, mouse-up, and mouse-dragged events are discrete types of the mouse events category. Some categories contain a single *compound event* whose event record contains information specifying which of several possible actions caused the event. The distinction between discrete and compound events will become clear from the examination of the event record later in this chapter.

Keyboard Events

Among the most common events sent to an application are these direct reports of the user's keyboard actions:

- A *key-down* event when the user generates a character by pressing a key
- A *key-up* event when the key is released
- A *flags-changed* event when the user presses or releases the Alternate, Shift, Control, or Command key, or turns Alpha Lock on or off

Of these, key-down events are the most useful to the application. Key-up events are less used since they follow almost automatically when there has been a key-down event.

Because the event record for every event type includes flags that indicate the state of Alpha Lock, Alternate, Shift, Control, and Command, applications normally don't need to receive flags-changed events; they're useful only for applications that have to keep track of the state of these keys continuously. The default event mask for a new window doesn't allow flags-changed events.

Mouse Events

Mouse events are generated by changes in the state of the mouse buttons and by changes in the position of the mouse cursor on the screen. Along with keyboard events, they're among the most useful to applications. This category consists of:

- Two sets of *mouse-down* and *mouse-up* events, one for the left mouse button and one for the right. "Mouse-down" means the user pressed the button; "mouse-up" means the button was released. If the mouse has just one button, only left mouse events are generated.
- Two types of *mouse-dragged* events—one for when the mouse is moved with its left mouse button down, or with both buttons down, and one for when it's moved with just the right button down. A mouse with a single button generates only left mouse-dragged events. As the mouse is moved with a button down, a series of mouse-dragged events is produced. The series is always preceded by a mouse-down event and followed by a mouse-up event.

- A *mouse-moved* event when the user moves the mouse without holding down either mouse button.
- *Mouse-entered* and *mouse-exited* events if the application has asked the Window Server to set a *tracking rectangle* in a window. These events report whether the cursor has entered the rectangle or left it. For each rectangle, you can specify whether these events should be generated only when one or another (or both) of the mouse buttons is being held down. A window can have any number of tracking rectangles; the event record identifies which rectangle was entered or exited. For more information on setting a tracking rectangle, see “Mouse-Exited and Mouse-Entered Events” in Chapter 7, “Program Dynamics.”

Mouse-dragged and mouse-moved events are generated repeatedly as long as the user keeps moving the mouse. If the user holds the mouse stationary, neither event is generated until it moves again. Nevertheless, mouse-dragged and mouse-moved events are different from all other events in that they report continuous rather than discrete user actions. They therefore place a heavy load on the system; applications should ask for them only when necessary. The default event mask for a newly created window doesn’t allow either of these two mouse events. You need to reset the mask when you’re ready to respond to these events and set it back again when you’re done. See “Modal Event Loops” in Chapter 7 for an example of how this can be done.

Note: The use of the terms “left” and “right” above is a matter of convention (see Chapter 2, “NeXT User Interface” for more information). Unless the mouse buttons have been differentiated using the Preferences program, either mouse button generates the “left” version of these mouse events. If, on the other hand, a user unlinks the two buttons, the primary mouse button (either left or right) generates “left” mouse events and the other button generates “right” mouse events.

Timer Events

A *timer event* notifies an application that a certain time interval has elapsed. An application can register that it wants timer events and that they should be placed in its event queue at a certain frequency. When the application no longer needs them, the flow of timer events can be turned off. An application can’t have more than one stream of timer events active at a time. These events are a service of the routines that place events in an application’s event queue and so aren’t created by the Window Server.

Timer events provide a way for applications to interleave two activities. For example, consider automatic scrolling. As the user drags the mouse outside a window, the application checks for mouse-dragged events and sends a message to select and scroll the contents of the window. The number of lines scrolled at a time depends on the distance the mouse is dragged away from the top or bottom edges of the window: The farther it’s dragged, the greater the number of lines scrolled at a time. However, the sending of scrolling messages can’t be contingent on the receipt of mouse-dragged events. If the user stops moving the mouse but continues to hold the mouse button down, the flow of mouse-dragged events stops although the window should continue scrolling.

To solve this problem, the application can specify that, while it's checking for mouse-dragged events (whether it finds one or not), it will receive timer events at a given interval. Each time it receives a timer event, it can send a message to the window to scroll.

Cursor-Update Events

A *cursor-update* event informs an application that the cursor has crossed the boundary of a predefined rectangular area within a window. The application can then respond by updating the cursor's shape. For example, the Edit application changes the cursor from an arrow to an I-beam whenever the cursor is within the text area of the window. There may be one or more rectangles defined at a time, and the rectangles can overlap partially or fully. A different cursor can be defined for each rectangle.

Cursor-update events may seem closely related to mouse-entered and mouse-exited events. In fact, the Application Kit converts specially marked mouse-entered and mouse-exited events into cursor-update events and then sends them on to the application. Consequently, these events are only available to applications based on the Kit. The Kit also provides the facilities for registering a cursor and assigning it to a particular area of a window. Once your application has assigned a cursor to a specific area, the Application Kit does the rest. Kit routines check for incoming cursor-update events and update the cursor's image accordingly. In doing so, the Kit consumes the cursor-update events, making it unnecessary for your code to handle them.

Kit-Defined Events

Kit-defined events contain information that the Application Kit uses to manage your application. Since the Kit is designed to respond to events of this type, the code you write will rarely need to take them into account.

A kit-defined event is a compound event, a single event type that serves as a vehicle for various subevents. By examining the event record of a kit-defined event, an application can determine which of several kit-defined subevents has occurred. These subevents report changes to a window or to the status of the application that created the window. Kit-defined subevents are:

- A *window-moved* subevent when the user has dragged the window to another location on the screen. Most applications won't care exactly where a user moves a window and won't need to respond to this subevent. The Application Kit uses it to keep track of where windows are on the screen.
- A *window-resized* subevent when the application has the Window Server resize a window in response to the user's actions. The subevent reports the window's new dimensions to the application. Although the Window Server resizes the window, it doesn't generate the window-resized subevent. Instead, the Application Kit creates the subevent and sends it on to its destination.

- A *window-exposed* subevent when a nonretained window that was covered or positioned off-screen becomes exposed. This subevent lets the application know which part of the window needs to be redrawn. Window-exposed subevents also occur for retained windows when instance drawing in them is exposed. (See “Instance Drawing,” in Chapter 4, “Drawing.”)
- An *application-activated* subevent when an inactive application is activated. This subevent alerts the application to display the markings on its main and key windows and to redisplay any panels that it hid when it became inactive.
- An *application-deactivated* subevent to cause a previously active application to become inactive. The application receiving an application-deactivated subevent makes its panels disappear and relinquishes the main and key windows.

System-Defined Events

A system-defined event is also a compound event, serving to group subevents which report activities of system-wide importance. As with kit-defined events, the Application Kit responds to system-defined events, making it unnecessary in most cases for you to handle these events directly.

Currently, there’s only one subevent, the *power-off* subevent, of the system-defined event. This subevent is generated when the user turns the power off by pressing the Power key on the keyboard. Before the power actually goes off, applications have time to notify the user of files not saved and do other cleanup if necessary. This subevent is not generated by power failures or any other means that might be used to turn the computer off.

Application-Defined Events

An application-defined event is a compound event with no predefined subevent types. An application can define as many subevent types as it needs. The event record contains several fields that the application can use to give specific content to the subevent.

An application might make use of these events in a number of ways. From the keyboard and mouse events it receives, it could recognize the user’s intentions and generate an event record for the appropriate application-defined subevent. It could post the event record to its queue and then process the record as it does those it receives from the Window Server. (See “The Event Queue” at the end of this chapter and **DPSPostEvent()** in *NeXTstep Reference, Volume 2*.)

The Event Record

Information about each event is stored in an *event record*. In some cases, the Application Kit extracts information out of the event record for you; usually, you'll need to access it directly. When the application sees it, the event record has the structure shown below. This structure is defined, along with event-related constants and macros, in the NeXT header file **dpsclient/event.h**.

```
typedef struct _NXEvent {
    int          type;           /* event type */
    NXPoint     location;       /* mouse location */
    long         time;           /* time since startup */
    int          flags;          /* key state flags */
    unsigned int window;        /* window number */
    NXEventData data;           /* type-specific information */
    DPSContext  ctxt;          /* context number */
} NXEvent;
```

Here's what the event record contains in its various components (called "fields" or "members" in standard C terminology):

Component	Content
type	The event type—whether it's a mouse-down, key-up, or kit-defined event, for example. (See the list in the next section.)
location	The mouse location—its x and y coordinates in the window's base coordinate system—immediately following the event, except for two subevents: <ul style="list-style-type: none">• For window-moved subevents, it's the lower left corner of the window after the move, in screen coordinates. (This point is the origin of the base coordinate system.)• For window-exposed subevents, it's the lower left corner of the rectangle to be redrawn, in base coordinates.
time	The time that the event occurred relative to system startup. The unit of time is hardware-dependent; on the NeXT computer, it's measured as the number of vertical retrace intervals (1/68 seconds) since the startup of the Window Server.
flags	Flags indicating whether the mouse buttons or modifier keys like Shift and Command were down when the event occurred, or whether a key that was pressed was on the numeric keypad (see "Event Flags" below).

window	In most cases, the window number of the window associated with the event. For left mouse-down, left mouse-up, and left mouse-dragged events, it's the window that the cursor was over when the mouse button was pressed. For right mouse events that occur over a window in the active application, it's also the window that the cursor was over when the mouse button was pressed. However, for right mouse events that occur elsewhere, it's the number of the window that last received a left mouse-down event. (The Application Kit uses this information to bring up the application's main menu in response to a right mouse-down event.) For mouse-moved events, it's the number of the topmost window that accepts this type of event.
	In the case of application-activated and application-deactivated subevents, this field identifies the application that's being deactivated (for application-activated subevents) or the application that's being activated (for application-deactivated subevents).
data	Type-specific information (see "Type-Specific Event Data" below).
ctxt	A number identifying the PostScript context associated with this event. Since an application typically creates only one context in the Window Server, the content of this field won't be of interest. This field is provided for applications that establish multiple PostScript contexts and that must know which context an event arrived from. (See "The Event Queue" at the end of this chapter and Chapter 4 for more information about PostScript contexts.)

Event Types

The event type is indicated by one of these symbolic constants:

Constant	Event Type
NX_KEYDOWN	Key-down
NX_KEYUP	Key-up
NX_FLAGSCHANGED	Flags-changed
NX_LMOUSEDOWN	Mouse-down, left or only mouse button
NX_LMOUSEUP	Mouse-up, left or only mouse button
NX_RMOUSEDOWN	Mouse-down, right mouse button
NX_RMOUSEUP	Mouse-up, right mouse button
NX_MOUSEMOVED	Mouse-moved
NX_LMOUSEDRAGGED	Mouse-dragged, left or only mouse button
NX_RMOUSEDRAGGED	Mouse-dragged, right mouse button
NX_MOUSEENTERED	Mouse-entered
NX_MOUSEEXITED	Mouse-exited
NX_TIMER	Timer
NX_CURSORUPDATE	Cursor-update
NX_KITDEFINED	Kit-defined
NX_SYSDEFINED	System-defined
NX_APPDEFINED	Application-defined

As a convenience, these synonyms for left mouse events are provided for your use:

Constant	Equivalent to
NX_MOUSEDOWN	NX_LMOUSEDOWN
NX_MOUSEUP	NX_LMOUSEUP
NX_MOUSEDRAGGED	NX_LMOUSEDRAGGED

Event Flags

The **flags** component of the event record contains the flags illustrated in Figure 5-1. Each flag is 1 in the situation stated in the diagram, and 0 otherwise. Bits that aren't labeled in the diagram are reserved for future use.

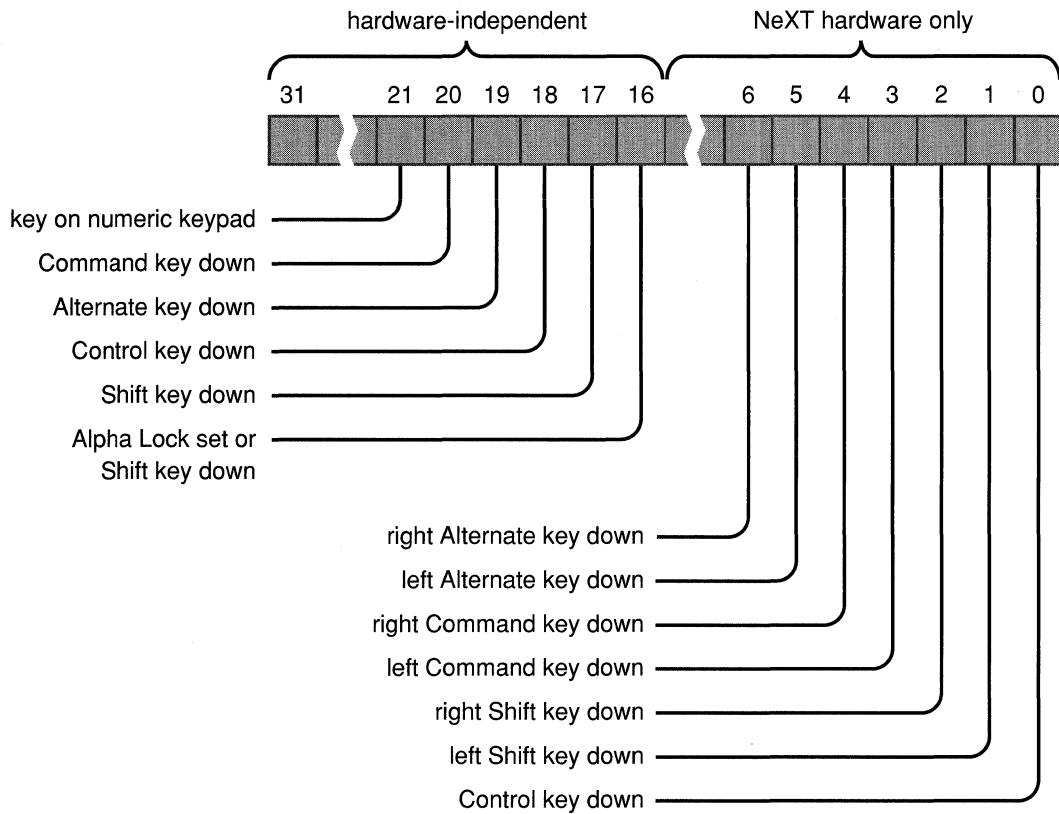


Figure 5-1. Event Flags

Only the flags in the high-order half of the flags integer are hardware-independent, that is, guaranteed to have the indicated meaning for any keyboard or mouse. The hardware-dependent flags (in the low-order half) should be used only when your application must distinguish between the left and right Alternate, Control, or Command keys. If you use the hardware-dependent flags, your application will require changes before it will work on later versions of the NeXT computer keyboard or with NeXT software running on other hardware systems.

The low-order event flags imply certain high-order flags. If either of the low-order flags corresponding to Alternate is set, the high-order Alternate flag is also set. Similarly, if either low-order flag referring to Command or Shift is set, the corresponding high-order bit is set. If the low-order Control flag is set, so too will be the high-order Control flag.

Note: The Command and Alternate keys may be labeled differently on other keyboards, and you may set Alpha Lock differently. Flags referring to keys entirely missing from other keyboards will always be 0 when those keyboards are used.

The following constants are provided as masks for accessing these flags.

Hardware-independent:

Mask	Event Flag
NX_ALPHASHIFTMASK	1 if Alpha Lock is set or any Shift key is down
NX_SHIFTMASK	1 if any Shift key is down
NX_CONTROLMASK	1 if any Control key is down
NX_ALTERNATEMASK	1 if any Alternate key is down
NX_COMMANDMASK	1 if any Command key is down
NX_NUMERICPADMASK	1 if key is on numeric keypad

NeXT hardware only:

Mask	Event Flag
NX_NEXTCTRLKEYMASK	1 if Control key is down
NX_NEXTLSHIFTKEYMASK	1 if left Shift key is down
NX_NEXTRSHIFTKEYMASK	1 if right Shift key is down
NX_NEXTCMDKEYMASK	1 if left Command key is down
NX_NEXTRCMDKEYMASK	1 if right Command key is down
NX_NEXTLALTKEYMASK	1 if left Alternate key is down
NX_NEXTRALTKEYMASK	1 if right Alternate key is down

For example, the following code tests whether Alternate was pressed at the time of the last event; **theEvent** is a pointer to the event record of the current event.

```
if ( theEvent->flags & NX_ALTERNATEMASK )
    doSomething();
else
    doSomethingElse();
```

Since the state of Alternate, Alpha Lock, Shift, and Control is reflected in the character information returned for a keyboard event (as described below), there's normally no need to check flags for them.

Type-Specific Event Data

The **data** component of the event record is a C union that contains different information depending on the type of event:

```
typedef union {

    /* For mouse-down and mouse-up events: */
    struct {
        short reserved;
        short eventNum;           /* event number */
        int click;                /* single, double, triple, etc.*/
        int unused;
    } mouse;
```

```

/* For key-down and key-up events: */
struct {
    short             reserved;
    short             repeat;      /* key repeat indicator */
    unsigned short    charSet;    /* character set */
    unsigned short    charCode;   /* character code */
    unsigned short    keyCode;    /* key code */
    short             keyData;    /* hardware-dependent data */
} key;

/* For mouse-entered and mouse-exited events: */
struct {
    short  reserved;
    short  eventNum;           /* mouse-down event number */
    int    trackingNum;        /* tracking rectangle number */
    int    userData;           /* programmer-determined data */
} tracking;

/* For kit-defined, sys-defined, and app-defined events */
struct {
    short  reserved;
    short  subtype;            /* subevent type for compound events */
    union {
        float  F[2];
        long   L[2];
        short  S[4];
        char   C[8];
    } misc;
} compound;
} NXEventData;

```

As shown here, **data** provides additional information for some types of events. For other types of events, **data** is undefined.

Mouse-Down and Mouse-Up Events

For a mouse-down or mouse-up event, **data.mouse.click** tells whether the event is part of a click, double-click, triple-click, or another number of successive clicks:

Value	Means
1	Mouse-down or mouse-up of a single-click in a window that is already the key window.
2	Mouse-down or mouse-up of a double-click
3	Mouse-down or mouse-up of a triple-click
<i>n</i>	<i>n</i> th successive click

For applications based on the Application Kit, **data.mouse.click** can also have a -1 value. This is made possible by PostScript procedures that the Application Kit loads into the Window Server when an application starts. Among the many functions of these procedures (called *packages*), is to assign a -1 value to the first click in a non-key window. The Application Kit then has the opportunity to interpret a first click differently from following clicks. For example, a first click in a non-key window containing text makes the window the key window but doesn't affect the selection in the text. A second click, however, moves the insertion point in the text to the location of the mouse event.

The Window Server interprets two or more clicks as either individual clicks or as part of a multiple-click, depending on the proximity of the clicks in time and location. You can modify the default time settings with the Preferences application.

Key-Down and Key-Up Events

When the user holds a repeating key down, the Window Server sends a succession of key-down events followed by a key-up event when the user releases the key. For a key-down or key-up event, **data.key.repeat** indicates whether the event is a discrete keyboard event or part of a succession of events generated by a repeating key. Keyboard events that are part of a succession are marked with a nonzero value in **data.key.repeat**; discrete events contain 0 in this field.

After sending the first key-down event, the Window Server waits a preset length of time before sending repeated key-down events. This delay, and the frequency of the repeated key-down events, can be set using the Preferences application.

data.key.charSet indicates the *character set* and **data.key.charCodeAt** indicates the *character code*, which together correspond to a particular character. The *key code*, **data.key.keyCode**, which maps to a key on the keyboard, will rarely be used. **data.key.keyData**, which can contain hardware-dependent data, is undefined for the NeXT keyboard. See "Keyboard Information" below for more information.

Mouse-Entered and Mouse-Exited Events

A mouse-entered or mouse-exited event's **data.tracking.trackingNum** field contains an arbitrary number that the application supplied when it set the tracking rectangle. The **data.tracking.eventNum** field contains the event number of the latest mouse-down event. The tracking number enables applications to associate mouse-exited and mouse-entered events with the correct tracking rectangle. If the mouse is being tracked only while one of its buttons is pressed, **eventNum** should match the event number of the mouse-down event that initiated the tracking.

You can use the **data.tracking.userData** field for your own purposes. It's commonly used to identify an object in your application that you want notified whenever the mouse enters or exits a tracking rectangle. Whereas the value in **data.tracking.trackingNum** must be unique for each tracking rectangle, any number of tracking rectangles can share the same **data.tracking.userData** value, which can be used to identify this object.

Kit-Defined Events

For a kit-defined event, **data.compound.subtype** identifies the kit-defined subevent. The constants corresponding to these subevents are:

Constant	Subevent Type
NX_WINMOVED	Window-moved
NX_WINRESIZED	Window-resized
NX_WINEPOSED	Window-exposed
NX_APPACT	Application-activated
NX_APPDEACT	Application-deactivated

A window-moved subevent's **location** field contains the window's new location in screen coordinates.

A window-resized subevent's **location** field contains the window's new location in screen coordinates. **data.compound.misc.L[0]** contains the window's new width, and **data.compound.misc.L[1]** contains the window's new height.

A window-exposed subevent's **location** field contains the location of the rectangle to be redrawn, given in base coordinates. **data.compound.misc.L[0]** and **data.compound.misc.L[1]** contain the width and height of the rectangle, also in base coordinates.

Application-activated and application-deactivated subevents are somewhat different from the other kit-defined subevents. Application-activated and application-deactivated subevents are sent in pairs. When a user clicks in the window of an inactive application, the currently active application receives an application-deactivated subevent, and then the application that received the click receives an application-activated subevent. For an application-deactivated subevent, the event record's **window** field contains the PostScript context number of the application that will receive the application-activated subevent. Similarly, for an application-activated subevent, the event's **window** field contains the PostScript context number of the application that received the application-deactivated subevent. If the application-activated event was caused by a mouse click in one of the application's windows, the event's **data.compound.misc.L[1]** field contains a number identifying the window the event occurred in.

System-Defined Events

A system-defined event currently has only one subevent type, the power-off subevent. For this system-defined subevent, **data.compound.subtype** is equal to the constant **NX_POWEROFF**. No other field of the event record contains useful data. All applications, including the Workspace Manager, receive this event. Those applications that were started from the Workspace ignore the event, waiting instead for notification from the Workspace Manager. Applications that were started from Terminal or Shell can respond to this subevent directly, if they choose. (See “System-Defined Events” in Chapter 7 for details about handling power-off subevents.)

Application-Defined Events

For an application-defined event, **data.compound.subtype** identifies the subtype of the event, as defined by the application. **data.compound.misc** has fields for each of the following data types: two floats, two longs, four shorts, and eight characters. An application can use these fields in any way it chooses.

Keyboard Information

For every keyboard event, the **data** component of the event record specifies the character that was typed, by indicating the *character set* and the *character code*. The character set is one of the following:

NX_ASCIISET	Standard (extension of ASCII)
NX_SYMBOLSET	Symbol

The standard character set is an extended version of ASCII (the American Standard Code for Information Interchange); this is the character set for most fonts. There’s a separate character set for the Symbol font. Each character set has an *encoding vector* that maps the character code to a particular character. The *NeXT Technical Summaries* manual shows the encoding vectors for each character set.

The character information in the event record incorporates the effect of Alternate, Shift, Control, and Alpha Lock (Command-Shift), but not necessarily Command (except for Command-Shift). For example, Command-Return (which is Enter) has a different character code than Return. On the other hand, D has the same character code as Command-D. Command-D is distinguished from D in that the event flag for Command is set in the event record.

The **flags** field can also be used to distinguish between the possible ways the codes for the arrow characters can be generated. If, for example, the code for the down arrow is generated by pressing the down arrow key, the numeric keypad flag is 1. Applications should interpret such a character code as a command to move the insertion point down one line in the text. If, on the other hand, the code is generated by pressing Alternate-Shift-n,

the numeric keypad flag is 0. Applications should respond to this by drawing a down arrow in the text. The Application Kit's Text class interprets these codes correctly; if your application implements its own text editor, it should test the numeric keypad bit to support this convention.

On future international versions of the NeXT keyboard, an individual key may correspond to a symbol formed by combining two different characters (for example, a letter and an accent). In this case, the application will receive a sequence of two key-down events followed by two key-up events, even though only a single key was pressed. If the key is held down, multiple key-down events will also be generated in pairs.

The **data** component of the event record also contains:

- A hardware-dependent key code, which indicates the position on the keyboard of the key that was pressed or released. The key codes for the NeXT keyboard are listed in *Technical Summaries*.
- Two additional bytes of hardware-dependent data; for events generated from the NeXT keyboard, these bytes contain 0.

Event Masks

Each window has an *event mask* maintained by the Window Server. The mask is a long integer with a bit assigned to each event type. For events coming from the Window Server, if the bit for a given type is 0, no events of that type will be associated with the window and sent to the application process. The Window Server passes an event to the window's application only if the event mask allows it.

When a window is created by the Window Server, it has a mask that allows all but mouse-moved, mouse-dragged, and flags-changed events. A standard window created with the Application Kit has an event mask that allows it to receive these events:

key-down	mouse-entered
key-up	mouse-exited
left mouse-down	kit-defined
left mouse-up	system-defined
right mouse-down	application-defined
right mouse-up	

Applications can set the event masks for the windows they create. You should be careful not to place an unnecessary burden on the Window Server by asking for events your application doesn't need. "Event Masks" in Chapter 7 tells how to change event masks through the Application Kit.

To help you set event masks and test event types, the NeXT header file **dpsclient/event.h** defines a mask for each event type. As shown below, the mask name is the same as the symbolic constant for the event type followed by “MASK”. There’s also a symbolic constant for a mask that accepts all types of events.

Constant	Event Type
NX_KEYDOWNMASK	Key-down
NX_KEYUPMASK	Key-up
NX_FLAGSCHANGEDMASK	Flags-changed
NX_LMOUSEDOWNMASK	Mouse-down, left or only mouse button
NX_LMOUSEUPMASK	Mouse-up, left or only mouse button
NX_RMOUSEDOWNMASK	Mouse-down, right mouse button
NX_RMOUSEUPMASK	Mouse-up, right mouse button
NX_MOUSEMOVEDMASK	Mouse-moved
NX_LMOUSEDRAGGEDMASK	Mouse-dragged, left or only mouse button
NX_RMOUSEDRAGGEDMASK	Mouse-dragged, right mouse button
NX_MOUSEENTEREDMASK	Mouse-entered
NX_MOUSEEXITEDMASK	Mouse-exited
NX_TIMERMASK	Timer
NX_CURSORUPDATEMASK	Cursor-update
NX_KITDEFINEDMASK	Kit-defined
NX_SYSDEFINEDMASK	System-defined
NX_APPDEFINEDMASK	Application-defined
NX_ALLEVENTS	All event types

There are some restrictions concerning setting the masks for timer and cursor-update events. As described earlier, timer events are created outside the Window Server, so setting the Server’s event mask to exclude this type has no effect. The NX_TIMERMASK constant is only used to check for timer event records in the event queue.

Cursor-update events are based on mouse-entered and mouse-exited events, and so setting a window’s mask to exclude these events could interfere with an application’s ability to update the cursor. To solve this problem, the Application Kit forces the key window to accept mouse-entered, mouse-exited, and cursor-update events if any cursors have been assigned to areas within the window. The Application Kit overrides the event mask only temporarily; when the need for tracking the cursor is removed, or the window is no longer the key window, the mask is reset to the last requested value. See “Changing the Cursor” in Chapter 7 for more information.

To convert a variable event type to a mask, you can use the macro NX_EVENTCODEMASK, also from **dpsclient/event.h**. For example, NX_EVENTCODEMASK(NX_KEYUP) is the same as NX_KEYUPMASK. If **someEvent** is an event record, NX_EVENTCODEMASK(someEvent.type) is a mask for its type.

When you want to create a mask of several types, use the bitwise OR operator to join together the masks for each type. Here a mask for keyboard events is used to test the type of **someEvent**:

```
NX_EVENTCODEMASK(someEvent.type) & (NX_KEYDOWNMASK | NX_KEYUPMASK)
```

Notice that each compound event type has only one mask: You can't set masks for individual subevents of a compound event, only for the compound event itself.

The Event Queue

Events are returned from the Window Server over a bidirectional communication channel. This is the same channel that the application uses to send drawing instructions to the Server. Chapter 4, discusses this interface to the Window Server and the various kinds of data that pass from the application to the Server. This section introduces some of the functions that access the data returned from the Window Server.

As the Window Server creates each event record, it's dispatched to the application process. When the event record arrives in the application process, it's added to the application's event queue. The event queue is a ring buffer that can hold a maximum of 50 events. The queue created and maintained by routines defined in the library **libNeXT.a**. This library includes the basic Display PostScript client library along with extensions that support the NeXT window system.

For applications based on the Application Kit, event records are taken from the queue and returned to the application whenever the main event loop is executing. If event records are consistently added to the queue faster than the application can remove them, the capacity of 50 records can be reached. Although this rarely happens, if it does a new event record replaces the oldest one in the queue.

The events the application finds are distributed by Kit objects and can be accessed by the custom objects you create. Your custom objects can also preempt the normal distribution process if necessary. (See Chapter 7 for more information on event handling in the Application Kit.) Since the Application Kit has a structure for accessing events and defines object-oriented interfaces for the event handling most applications need, in most cases programmers using the Kit shouldn't make direct calls to the underlying event-handling functions.

Applications that don't use the Application Kit must access the event queue directly. The functions that check the event queue are **DPSGetEvent()** and **DPSPeekEvent()**.

DPSGetEvent() retrieves a specific type of event from the queue; **DPSPeekEvent()** simply checks to see if a specific type of event is present. Each function has a *timeout* argument that determines how long it will wait for an event before returning. **DPSDiscardEvents()** removes an event of a specific type from the event queue. An application can add an event to the beginning or end of its own queue using **DPSPostEvent()**.

Two low-level functions that might be used by both Kit and non-Kit applications are **DPSSetEventFunc()** and **DPSSetTracking()**. **DPSSetEventFunc()** lets you establish an event filter for a particular context. When the Window Server sends an event to the application, it is examined by the filter function. Depending on the filter function's return value, the event is then either discarded or added to the event queue. This allows your application to single out certain events for special processing before letting them enter the event queue. **DPSSetTracking()** affects how multiple mouse-dragged and mouse-moved are treated in the event queue. As an optimization, a string of mouse-dragged or mouse-moved event records can be coalesced into a single event record. For example, the event queue can fill with numerous mouse-dragged events when the user drags the mouse rapidly. Rather than process each event separately, applications by default coalesce these events into a smaller number of similar events. **DPSSetTracking()** lets you turn off this behavior in case it's important for your application to receive each event. A drawing program, for instance, might need each discrete event so that it can accurately follow the trajectory of the user's mouse movements. Note, however, that by turning off event coalescence, your application runs a greater risk of overrunning the capacity of the event queue.

See *NeXTstep Reference, Volume 2* for descriptions of each of the functions mentioned above.

Event-Related Services

The client library routines that manage the event queue can also provide other services to an application. They can:

- Execute a timed entry.
- Monitor a Mach port for messages.
- Monitor a file descriptor for data.

An application can request these services by calling the appropriate function:

- **DPSAddTimedEntry()**
- **DPSAddPort()**
- **DPSAddFD()**

For each service, you register a function that you want called when the corresponding condition is met. For example, **DPSAddPort()** requires a function to call when a message arrives at a Mach port. The function is called only if there's work for it to do—the timed entry must be due, a message must have been received at the port, data must be ready to read from the file.

The registered functions take different arguments depending on which service has been requested. However, they have an argument in common: one of type **void ***. You can use this generic pointer to pass information to the function. For example, you might use it to pass the **id** of an object within your application that the function should send a message to.

When an application requests one of these services, it must also assign it a priority level. Before a condition is checked, its priority level is compared to a threshold assigned by the application. The condition is checked (and the registered function called) only if its priority is equal or greater than the threshold. By temporarily raising the threshold, an application can defer processing input from other sources in favor of processing events.

The sections below describe these features in greater detail. For more information on the functions mentioned in the following sections, see the descriptions in *NeXTstep Reference, Volume 2*.

Executing Timed Entries

A *timed entry* is a function you specify to be called at a given frequency. If your application includes a timed entry, the system checks the entry's status (assuming its priority level qualifies it) whenever the application accesses the event queue. If the specified period of dormancy has elapsed, the timed entry is executed.

Applications can use timed entries for various ends. For example, the Application Kit's Text class employs a timed entry to blink the cursor. The Application Kit also uses a timed entry to insert timer event records into the requesting application's event queue.

You register a function as a timed entry by calling **DPSAddTimedEntry()**. The arguments to **DPSAddTimedEntry()** are the function to be called, the time interval between calls, an arbitrary piece of data to be passed to the function, and a priority level. Since you can have more than one function registered as a timed entry within an application, each call to **DPSAddTimedEntry()** returns a unique registration number. When a specific timed entry is no longer needed, it's eliminated by calling **DPSRemoveTimedEntry()** with the timed entry's registration number.

For example, consider a simple function that prints a character that's passed to it. To be used as a timed entry, the function would need a declaration of this type:

```
void printIt(DPSTimedEntry teNum, double now, void *userData)
{
    char *theChar = userData;
    printf("%c\n", *theChar);
}
```

The first two arguments aren't used in this example but could be used to identify the caller and the time of the call. The **userData** argument is a generic pointer that's serves to pass the character to the timed entry.

To register the function as a timed entry, you call **DPSAddTimedEntry()**, passing the interval, timed entry function, some arbitrary data (in this case, the character to be printed), and a priority level:

```
DPSTimedEntry teNum;
char aChar = '.';
double interval = 0.5;
int priority = 1;

teNum = DPSAddTimedEntry(interval, printIt, &aChar, priority);
```

(Priority levels are discussed in more detail in “Scheduling” below.)

After the timed entry is registered, the **printIt()** function is called every half second. Each time it's called, it prints the character pointed to by its third argument. Later, you could remove the timed entry by calling **DPSRemoveTimedEntry()**:

```
DPSRemoveTimedEntry(teNum);
```

Interval Variability

It's important to realize that the interval you set for the timed entry is only a request to the system; the actual interval depends on several factors. Foremost among them is that a timed entry is called only when the application is waiting for events. If one of your program's routines involves a long computation, no timed entry will be executed until the routine finishes the computation and returns control to the main event loop.

Once an application is free to check for a timed entry, several other factors come into play. It's obviously futile to specify a period of 1 second for a function that, when called, has an execution duration of 2 seconds. Less obvious is the problem caused by the interaction between multiple timed entries. If your application involves two timed entries, one of short duration and period and another of long duration, the execution of the first timed entry will be periodically delayed by the execution of the second. As the number of timed entries increases, so too does the possibility for interference among them. In a similar fashion, the execution of a timed entry can be delayed if the application is also processing a message at a Mach port or data at a file descriptor.

Another factor affecting the frequency of timed-entry execution is the resolution of the system clock. A timed entry having a period approaching the clock's resolution will experience a proportionately greater variability in its true period than a timed entry having a much longer period.

Finally, because your application is running within a multitasking operating environment, it's periodically suspended to allow execution time for other processes. When your application restarts, a timed entry may be overdue for execution. For some purposes, such as blinking an image, process suspension is of little consequence. For others, such as the simulation of the movement of a real object, you need to design the timed entry in a more sophisticated way to compensate for periodic process suspension. One strategy for

simulations of this type is to design the timed entry so that it calculates the state of the simulated object relative to the absolute system time, which, as shown in the example above, is passed as an argument to the function.

For example, consider two designs for a timed entry that draws the path of an object moving at a constant speed. In the first case, when the timed entry is called, it simply increments the object's coordinates and redraws the object. This strategy doesn't take into account that the application process will be suspended periodically during its execution. Consequently, the object's position as a function of time won't be accurate. In the second case, each time the timed entry is called, it checks the system time and determines the elapsed time since the start of the simulation. It then calculates the object's displacement from the starting point and draws the object in this new position. Although the object's motion will not be entirely smooth, this technique preserves the relationship between time and distance.

Checking Mach Ports

An application can use a Mach port as a conduit for communication with other applications. (For information on Mach and Mach ports, see the *NexT Operating System Software* manual.) Applications receive messages of various types at Mach ports; for example, when the Workspace Manager needs an application to open a file, it sends an appropriate message to one of the application's Mach ports.

The function **DPSAddPort()** adds a port to the list of ports that the application can check when it's getting events. It also registers a function to be called when a message is received. Other arguments are the message's maximum size, a pointer to user-defined data, and a priority level. Again, the function call is contingent on the function's assigned priority level. When an application no longer needs this service it calls **DPSRemovePort()**.

You'll rarely need to manage Mach ports and messages directly; the Application Kit furnishes a more convenient object-oriented interface to these services. It provides applications with the Mach ports they need to communicate with the Workspace Manager and with other applications. The Application Kit also defines classes that you can use to create and monitor any additional ports your application may require.

Checking File Descriptors

Applications can also communicate by sending data to, or receiving it from, a file. The file (which may represent a disk file, a socket, a device, or some other file type) is represented by an integer called a file descriptor. **DPSAddFD()** adds a file descriptor to the list of those the application checks (depending on priority level) each time it attempts to retrieve an event. Other arguments specify a function to be called to read the data from the file, a pointer to some user-defined data to be passed to this function, and a priority level.

Most UNIX commands are designed to take their input from one file and write their output to another. In addition, peripheral devices such as terminals and printers are represented by files. Given the design of the UNIX operating system, checking file descriptors provides your application with a general interface to other programs and devices. For example, your application could monitor a file descriptor to check for error messages generated during the compilation of another program.

Scheduling

Whether a timed entry is called, or a port or file descriptor is checked, depends largely on the priority level assigned to the registered function. The priority level is an integer from 0 to 30, with 30 as the highest possible priority and 0 as the lowest. The priority of a function is set when it's first registered (with the **DPSAddTimedEntry()**, **DPSAddPort()**, and **DPSAddFD()** functions).

Whenever an application checks for an event, it specifies a priority threshold. Registered functions with priorities lower than the threshold are screened out; all those with equal or higher priorities are checked at least once to see whether they should be called. If an application has to wait for an event to arrive in the queue, it's possible for a function to be checked and called many times over. Even if there's no wait, all functions at or above the threshold will be checked once.

In general, applications ask for events at a threshold of 1, which ensures that functions registered with a priority level of 1 or above will be called. The highest threshold an application can specify is 31, and the highest priority level for a function is 30. Thus, specifying a threshold of 31 prevents any registered function from being called, without regard for its priority level. Assigning a function a priority level of 0 effectively blocks it from ever being called, without regard for the current threshold. For more information on the use of thresholds in the Application Kit, see "Scheduling" in Chapter 7.

Note: A function's priority level is compared only to the threshold, not to the priorities of other functions. If two functions have priorities at or above the threshold, it doesn't matter that one may have a priority of 29 and the other a priority of 10. Both will be scheduled equally.

Chapter 6

Program Structure

6-4	Writing a Program with the Application Kit
6-4	Class Definitions
6-6	Using Kit-Defined Classes
6-7	Instances
6-7	Subclasses
6-7	Categories
6-8	Delegates
6-8	Structuring an Application
6-10	Responding to Kit-Generated Messages
6-12	Principal Application Kit Classes
6-14	Responder
6-15	View
6-16	The View Hierarchy
6-17	The Frame Rectangle
6-21	Drawing Coordinates
6-23	The Bounds Rectangle
6-24	The Visible Rectangle
6-24	Displaying a View
6-25	Associating Events with Views
6-26	View Subclasses
6-26	Control
6-27	Cell
6-28	Text and Icons
6-28	ActionCells
6-29	Window
6-29	Frame and Content Rectangles
6-30	Managing the View Hierarchy
6-30	Frame and Content Views
6-31	Coordinate Systems
6-31	Events
6-32	Window Subclasses
6-33	Application
6-33	Receiving Events
6-33	Window Management
6-34	Listener's Delegate
6-34	Global Services

6-36	Application-Specific Code
6-37	Overseeing Other Objects
6-37	Action Messages
6-37	Remote Messages
6-37	Application-Defined Events
6-37	Program Framework
6-38	The Core Framework
6-40	windowList
6-41	contentView
6-41	Superview and Subviews
6-42	window
6-43	nextResponder
6-44	Outlets
6-45	Delegates
6-46	Delegated Messages
6-48	Targets
6-49	Defining an Outlet
6-49	Named Objects
6-50	Managing Windows
6-50	Setting Up a Window
6-52	The Miniwindow
6-53	The Title
6-54	Changing the Close Button
6-54	Background Color
6-54	Deferred and One-Shot Windows
6-56	Hiding Panels
6-56	Window Status
6-57	Physical Management
6-57	Moving a Window
6-59	Resizing a Window
6-60	Reordering a Window
6-61	Getting Information about a Window
6-62	Frame and Content Rectangles
6-62	Window Numbers
6-62	Screen List Information
6-63	Environmental Information
6-64	Command-Line Arguments
6-66	Segments
6-68	Application Kit Conventions
6-68	Reading and Writing Instance Variables
6-68	Writing Methods That Create Instances
6-70	Defining a Subclass
6-71	Covering Inherited Methods
6-71	Returning self
6-71	Tags

Chapter 6

Program Structure

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

**/NextLibrary/Documentation/NextDev/ReleaseNotes/AppKit.rtf
/NextLibrary/Documentation/NextDev/ReleaseNotes/AllocInitAndNew.rtf
/NextLibrary/Documentation/NextDev/ReleaseNotes/Performance.rtf**

The simplest way to program an application for the NeXT computer is to take advantage of the facilities built into the three software kits—the Application Kit, Sound Kit, and Music Kit. The Application Kit provides a basic program structure for applications that draw on the screen and respond to events—for almost all NeXT applications, even those that mainly use the other kits. It implements the NeXT user interface and relieves you of many of the more tedious programming tasks common to most applications. When you use the Application Kit, you need to write only the code that makes your program distinct from a generic application.

Through the Application Kit, your program can:

- Open and maintain a connection to the Window Server.
- Place windows on the screen.
- Draw within the windows.
- Process keyboard and mouse events.
- Create buttons, menus, and the other graphical objects defined in the user interface.
- Display editable text.
- Print documents and on-screen displays.
- Write data to and retrieve data from the pasteboard.
- Manage off-screen bitmaps, including bitmaps for the cursor.
- Communicate with cooperating applications.

This chapter introduces the Application Kit and the program structure it defines. Chapter 7, “Program Dynamics,” follows with a more detailed discussion of drawing and event handling within this structure. The Sound Kit and Music Kit are discussed in *Sound, Music, and Signal Processing*.

The three software kits provide an integrated object-oriented programming environment. They’re coded in Objective-C, an object-oriented extension of the C language; programs that make use of the kits must also be written in Objective-C. However, through Interface Builder, you can develop much of your program graphically, on-screen, without writing any Objective-C code. Interface Builder constructs and initializes the user-interface objects for your application and sets up their connections to the rest of the program.

Chapter 8, “Interface Builder,” assumes the program structure that’s defined in the Application Kit and described in this and the following chapters. Chapter 9, “User-Interface Objects,” has a more detailed discussion of Kit-defined objects that draw and respond to events; these are objects that you can program using Interface Builder. Chapter 10, “Support Objects and Functions,” then presents the objects, functions, and macros that support the main functionality of your program.

If you’re unfamiliar with object-oriented programming and the terminology and syntax of Objective-C, consult Chapter 3, “Object-Oriented Programming and Objective-C.”

Writing a Program with the Application Kit

Every application consists of a network of objects—many defined in the Application Kit, some defined in the application you write. The structure of the network and the kinds of messages that get passed from object to object are the subjects of this and the following chapter.

Class Definitions

The Application Kit consists mainly of class definitions. It defines a separate class for each of the canonical control objects in the NeXT user interface. It also defines classes for panels and menus, for objects that permit users to enter and edit text, for an object that can manage the pasteboard for your application, for bitmaps and cursors, and for an object that oversees the entire application. All these classes, and others, are described in detail in this and the following chapters.

Like all Objective-C classes, the classes defined in the Application Kit are linked, through their superclasses, to an *inheritance hierarchy* that has the Object class at its root. Figure 6-1 shows the classes in the Application Kit inheritance hierarchy.

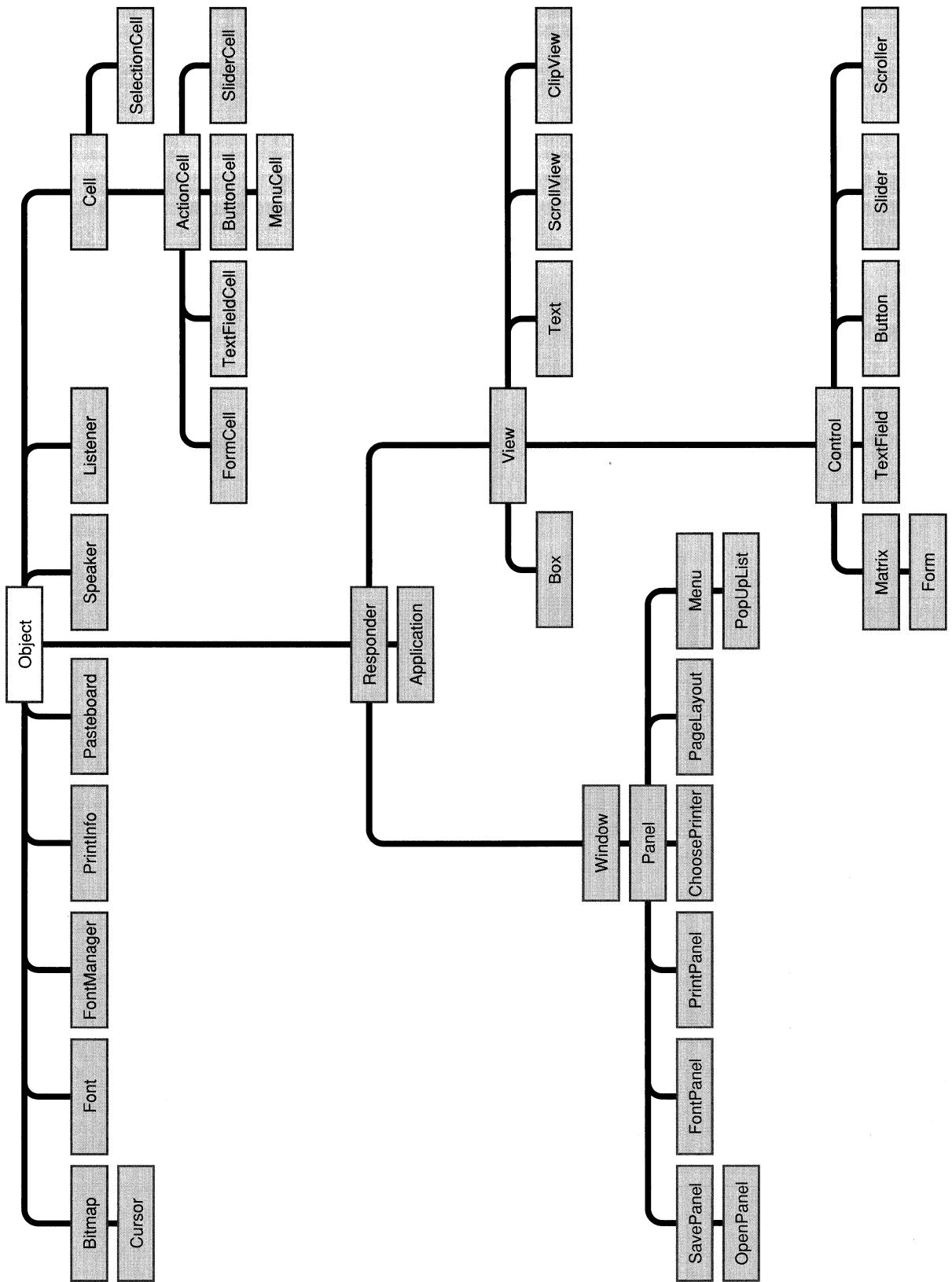


Figure 6-1. Application Kit Inheritance Hierarchy

This diagram shows, for example, that the Panel class has the Window class as its superclass, that the Window class has the Responder class as its superclass, and that the superclass of Responder is Object.

Each class inherits both instance variables (data structures) and methods (procedures) from all the classes above it in the hierarchy. The Panel class inherits from Window, Responder, and Object. This means that an instance of the Panel class can use not only Panel methods, but also methods defined in the Window, Responder, and Object classes. It also means that every Panel object has all the instance variables declared in the three classes it inherits from, in addition to any that might be declared in the Panel class itself.

Since the methods and instance variables that are defined in a class are inherited by its subclasses, a subclass can be viewed as a variant of the classes above it in the hierarchy. Inheritance makes Panel a kind of Window, and Window a kind of Responder. When we speak of a “Window object,” we mean not only objects belonging directly to the Window class, but also any object that belongs to any class that inherits from Window. Panels, OpenPanels, FontPanels, Menus, and PopUpLists are all Windows.

Using Kit-Defined Classes

You can make use of Application Kit classes in four ways:

- You can create objects (instances) belonging to the classes defined in the Kit. Your application can include Button, Slider, and Menu instances, for example.
- You can define new subclasses of Application Kit classes, then create instances of the subclasses for your application. Through its superclass, each class you define will inherit methods and instance variables from the Kit. The instance variables and methods you add in the subclass definition serve to adapt generic Kit capabilities to the specific needs of your application.
- You can add new methods to a Kit class by putting them in a category definition that extends the original definition of the class.
- You can define your own objects to act on behalf of objects that inherit from the Application Kit. Kit objects handle most of the work themselves, but can delegate responsibility for some of the messages they receive to other objects.

Instances

In many cases, creating an instance of an Application Kit class is all that's needed. Class definitions in the Kit give you a good deal of freedom to adjust an instance to the needs of your application. A Button, for example, can be shaped, titled, associated with a keystroke, and assigned an action of your choosing, simply by initializing its instance variables. (For a Button, you'd usually do this through Interface Builder.)

Subclasses

Since your application will do some things no other application will do, it will probably need to define some objects of its own. In most cases, these objects will be adaptations of objects defined in the Application Kit. For example, the Panel class defines a kind of window that behaves as a panel should according to the user-interface guidelines. If you need a panel with a particular display and a special relationship to the other windows in your application, you can define it as a subclass of Panel.

The main reason for defining subclasses of Kit classes is to design objects that draw and respond to events in ways that are specific to your application. Generally, you'd define subclasses of View.

A View is an object that has an area and a coordinate system for drawing within a window. It can be selected to receive keyboard and mouse events, and can be scrolled and automatically displayed. If you need an object with these features, you'd simply define a subclass of View and implement methods that draw and respond to events as you'd like. You wouldn't have to re-implement the methods that manage the object's coordinate system, display it on the screen, permit it to be scrolled, or select it to receive mouse and keyboard events. The subclass inherits all the methods and instance variables of the View class; you need to implement only the features you want to add or change.

You might also choose to define subclasses of specific kinds of Views. For example, the Button class defines one- and two-state buttons. If you want a button that can display three or more states, you can subclass Button to add the functionality that you need.

Categories

In simple cases, where the only modification you need to make to a Kit-defined class is the addition of some methods, you can add them in a category definition rather than in a new subclass. Categories extend the definition of an existing class to encompass new methods. They're discussed under "Adding to a Class" in Chapter 3.

Delegates

Some Application Kit objects permit other objects to intervene and control some of their actions. Because the Kit object delegates responsibility for its behavior to the controlling object, the controlling object is known as a *delegate*. A delegate does only as much as you program it to do (in its class definition). One delegate can serve a number of different client objects.

By centralizing application-specific code in the delegate, you can adapt the behavior of Application Kit objects without defining Kit subclasses. The delegate can be a subclass of an Application Kit class, but usually it's just a subclass of the generic Object class.

The section on “<The Extended Framework>,” later in this chapter, has more information on delegates and the messages they can receive.

Structuring an Application

When it's launched, an application creates a set of objects for itself, displays itself on the screen and waits for events from the Window Server. Whenever it receives an event, it initiates a message to the responsible object, then waits for the next event. The object that receives the message will initiate some messages of its own, so several objects are likely to be involved in the application's response to the event.

The response to an event always includes a visible reaction on-screen, if only to let the user know the event was received. Applications draw both to present the user with a display that can prompt for events and to show the results of those events. The user, the application, and the Window Server participate in a continuous cycle of action and response that's illustrated in Figure 6-2 below.

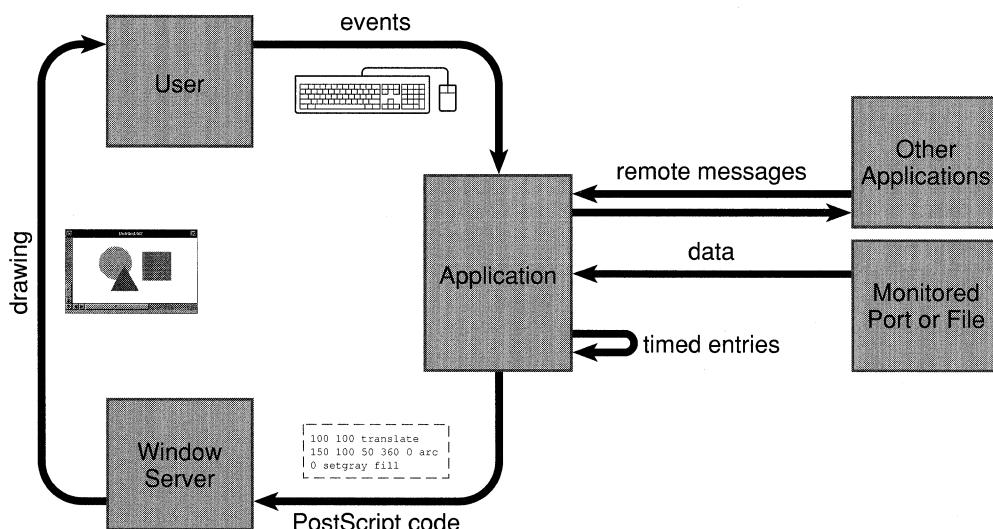


Figure 6-2. The Event Cycle

This cycle is driven by the user's actions. From the point of view of an application, it's driven by events. Between events, applications can respond to three other kinds of input:

- *Remote messages*, messages received from other applications. For example, an application might receive a message asking it to supply some data in text form to a word processor, or it might get a message from the Workspace Manager requesting it to open another file. Remote messages are handled just like messages initiated within the application itself.
- *Timed entries*, procedures that are to be called periodically and have been registered with the **DPSAddTimedEntry()** function.
- Data received at a monitored port or file. For example, an application might monitor changes to a central database (perhaps of messages written to the console). The monitoring procedure is registered using the **DPSAddPort()** or **DPSAddFD()** function.

Most everything that an application does is, directly or indirectly, a response to an event, to a remote message (which is itself prompted by an event in another application), to a timed entry (which is usually registered because of an event), or to data received at a monitored port or file (which changes due to user activity elsewhere).

To respond to events and remote messages, every application needs:

- An Application object, to get events from the Window Server
- Window objects, to provide areas on-screen where the application can draw
- View objects, to draw within the Windows and to handle keyboard and mouse events associated with windows
- A Listener, to receive remote messages, and a Speaker, to send them

The Application Kit creates a default Listener and Speaker for every application. You must create the Application object, Windows, and Views for yourself.

The Application object also oversees the application's Windows, and is usually expected to respond to the remote messages received by the Listener. Windows oversee their Views. All the objects are designed to work together in a coherent program structure.

This brief outline of how an application works is explained and expanded throughout the rest of this chapter.

Responding to Kit-Generated Messages

As it receives events from the Window Server and reacts to the instructions in your program, the Application Kit initiates a variety of messages. Very often, the message is designed to give the objects in your program the opportunity to do their own work in their own way. To take advantage of this opportunity, you need only implement the appropriate method in a class definition.

The chart below lists some of the methods that you might consider implementing. All these methods, and their place in the Application Kit's program structure, are discussed in detail later in this and the next chapter, and in Chapter 9. The chart is an introduction and for quick reference only.

If You Want to:	Write Your Own Version of:
Respond to events and subevents	keyDown: keyUp: flagsChanged: mouseDown: mouseUp: mouseDragged: mouseEntered: mouseExited: mouseMoved: windowMoved: windowExposed: windowResized: applicationDefined: powerOff:
Determine which events to receive	acceptsFirstResponder acceptsFirstMouse
Reflect change of event-handling status	becomeActiveApp resignActiveApp becomeKeyWindow resignKeyWindow becomeMainWindow resignMainWindow becomeFirstResponder resignFirstResponder
Respond to a keyboard alternative	performKeyEquivalent: commandKey:
Draw an object on the screen	drawSelf::

Open a file	validateFilename openFile:ok: appAcceptsAnotherFile appOpenFile:type:
Archive and unarchive an object	read: write: finishUnarchiving awake
Change the cursor	resetCursorRects
React when the user logs out	powerOffIn:andSave: appPowerOff:
Control text displays	textWillChange: textDidChange: textWillEnd: textDidEnd:endChar: textWillResize: textDidResize:oldBounds:invalid: textWillSetSel:toFont: textWillConvert:fromFont:toFont: textWillStartReadingRichText: textWillReadRichText:stream:atPosition: textWillFinishReadingRichText: textDidRead:paperSize: textWillWrite:paperSize: textWillWriteRichText:stream:forRun: atPosition:emitDefaultRichText: textIsEmpty:
Control a window	windowWillClose: windowWillReturnFieldEditor:toObject: windowWillMinimize:toMiniwindow: windowDidMinimize: windowDidDeminiaturize: windowWillResize:toSize: windowDidResize: windowDidExpose: windowDidMove: windowDidBecomeKey: windowDidResignKey: windowDidBecomeMain: windowDidResignMain: windowDidUpdate:

Manage the application	appDidHide: appDidUnhide: appDidInit: appDidBecomeActive: appDidResignActive: appDidUpdate:
Update a menu item	A method with a name of your own choosing
Respond to user actions on a control	A method with a name of your own choosing

Although you may write your own versions of these methods, you never send a message to have an object perform them. The message is initiated by the Application Kit as part of its procedures for handling an event, or in reaction to another message your program sends.

There's no penalty for failing to write your own versions of these methods. In a few cases (**textWillEnd:**, for example), the message won't be sent unless the receiving object has a method that can respond. In other cases (**mouseDown:** and **drawSelf::**, for example), the Kit defines a default version of the method that the receiving object can inherit and use.

Note: When defining a class, you're free to override any inherited method. However, most methods do what you need them to do, so there's no reason to override them (just as there's no reason to override functions in the standard C library). The difference between most methods and the methods in the list above is that the listed methods are designed to be overridden; most methods are not.

Principal Application Kit Classes

Application Kit objects are designed to work together. Through them, and through your own subclass adaptations of them, your program assumes a framework for dealing with events, drawing on the screen, and managing user-interface objects.

The inheritance hierarchy of Application Kit classes was presented in Figure 6-1 above. As that illustration shows, most Application Kit classes inherit from the Responder class, and almost all Responders also inherit from Window or View. The basic structure of a complete Application Kit program can be outlined by looking at just these three classes and the other Responder subclass, Application. Although other Kit classes—notably Panel, Menu, Listener, and Speaker—play important roles, the fundamental framework of an application is defined by Responder and its three subclasses.

Two other Kit classes, Control and Cell, make a vital enough contribution to program design to also be considered principal classes. The principal Application Kit classes are highlighted in Figure 6-3 on the next page.

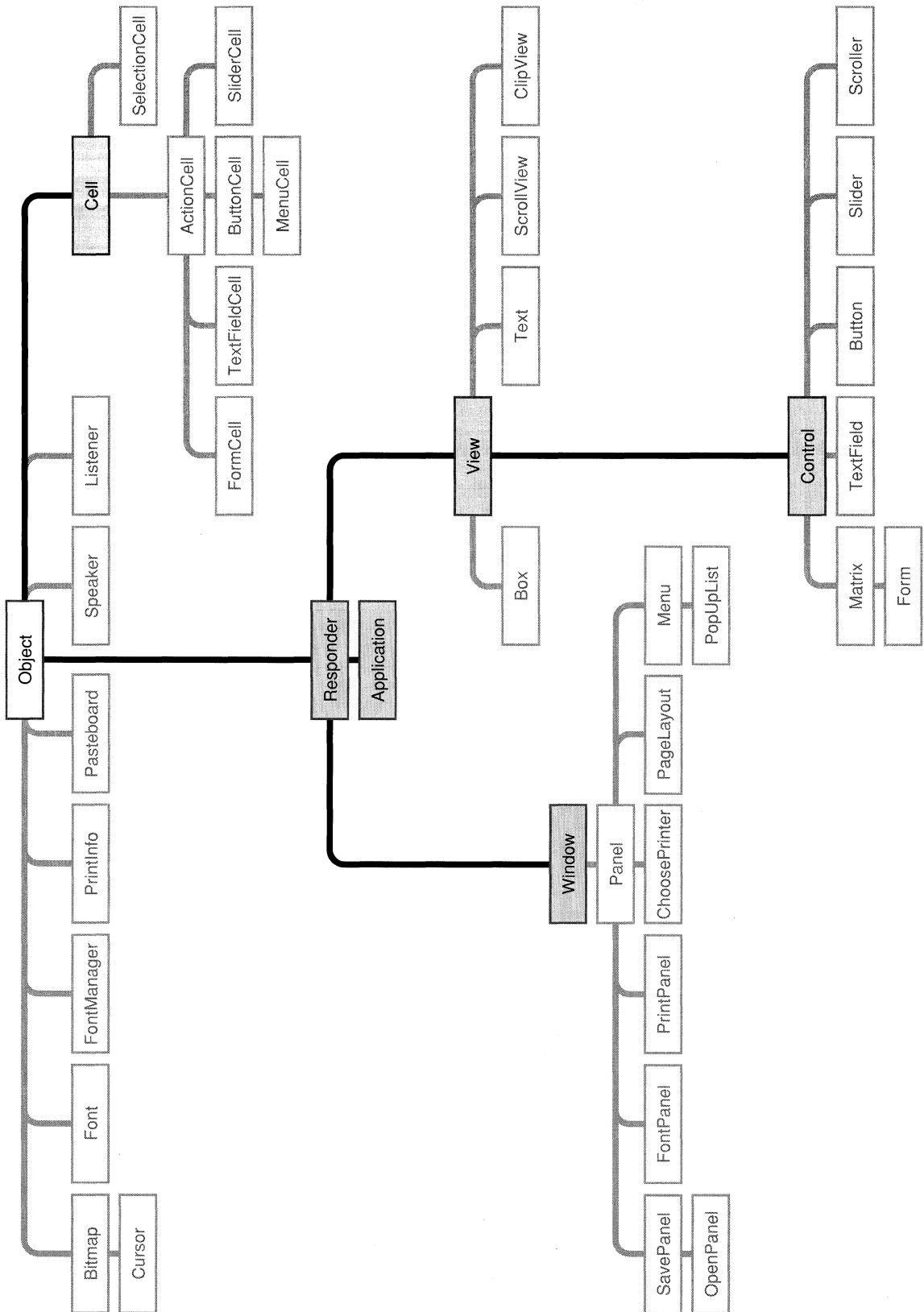


Figure 6-3. Principal Application Kit Classes

Responder is an abstract superclass; programs use instances of its subclasses rather than of Responder itself. Objects that inherit from Responder can, as its name implies, respond to events. Those that inherit from View can also draw on the screen. Since the mouse requires a graphical object for the cursor to point to, Views are the principal handlers of mouse events. Since typing must be displayed on-screen, Views also handle keyboard events. Most of the objects in your program will likely be Views of one sort or another.

Control is the abstract superclass for View objects that fit the control-action and target-selection paradigms of the user interface. The main attribute that Control imparts to its subclasses is the ability to translate events into application-specific messages for other objects. Each of the controls described in the user interface—buttons, sliders, text fields, and scrollers—is implemented as a separate Control subclass.

The Cell class defines an object that is very much like a View; it can draw and respond to events. But Cells lack the View superstructure that assigns the object an actual location within a window and permits it to receive events. Cells therefore must serve at the pleasure of Views. A View gets events for its Cells and tells them where and when to draw. Cells are a way of dividing a View into semi-autonomous regions, just as Views divide up the content area of a window. Most Control objects use a Cell to handle their drawing tasks.

Each Window object corresponds to a separate window provided by the Window Server. Every program will have at least one Window object, not only to open an area on the screen for the application to draw in, but also to supervise drawing and event handling among the View objects that are displayed within the window. Each Window typically has a number of Views arranged hierarchically inside it. Window's subclass, Panel, defines an object that can fill the role of a panel in the user interface. Panel's main subclass, Menu, defines objects that serve as menus.

Every program must also include one, and only one, Application object to act as its contact with the Window Server. The Application object supervises the entire program, receiving events from the Server and dispatching them to the appropriate Window objects. It manages the Windows in the application much as each Window object manages its Views.

The following sections amplify on this brief overview of these basic Application Kit classes. Other classes are described later in the chapter and in Chapters 9 and 10 as the Kit is examined in more detail. In addition, the *NeXTstep Reference* manuals describe every public method and instance variable that's defined in all the classes.

Responder

Responder is the abstract superclass that most other Application Kit classes inherit from. Those that don't inherit from Responder principally act as support for those that do. The objects that inherit from Responder respond to events; they receive the messages that are initiated by the user's actions on the keyboard and mouse.

Responders participate in a linked list of event-handling objects called a *responder chain*. If an object in the chain can't handle a message that's sent to it, the message is passed on to its *next responder*. If the next responder can't handle it either, the message continues to be passed up the chain from object to object in search of a Responder that can. This allows event handling to be consolidated in an object that collects messages originally intended for other Responders. The chain is unidirectional; messages are passed in one direction only.

The Responder class defines the elements essential to the responder chain, a **nextResponder** instance variable and the methods for passing messages from one object to another.

Only two types of message are passed along the chain. Both types are generated in direct response to the user's actions. They are:

- Messages that announce an event, such as messages to perform a **keyDown:** or **mouseExited:** method. This type of message is discussed under "Event Messages" in the "Event Handling" section of the next chapter.
- Messages that announce a user action on a Control object, such as a Button or a Slider. These messages are discussed under "Action Messages" in the "Event Handling" section of the next chapter.

Although Responder's contribution to event handling is crucial, it's also quite small. Its subclasses—especially Application, View, Window, and Control—add much more specific event-handling capabilities.

View

The View class provides a structure for drawing on the screen and for handling mouse and keyboard events. All the graphical objects defined in the NeXT user interface inherit from View. Views draw scrollers and buttons, display text, and even draw the borders and title bars of windows. The graphical objects you design must also be Views.

The drawing a View object places on the screen can be thought of as a visual representation of the object itself. In many cases, View subclasses are named for the objects they draw. A Button, for example, draws an image of a button that the cursor can point to and the user can click or press. We speak both of the Button object "drawing" and of it "being drawn" on the screen. In a sense, it draws itself.

Although all View objects are also Responders, you can define a View that draws but doesn't respond to events. Any events it happens to receive will be passed to another object through the responder chain.

The View Hierarchy

Every View object is associated with the particular window where it's displayed. All the Views within a window are linked together in a *view hierarchy*. Each View has another View as its *superview* and may be the superview for any number of *subviews*.

The view hierarchy benefits drawing in two ways:

- It permits a View object to be constructed out of other Views. For example, a graphical keypad might be a View with separate subviews for each key. A spreadsheet could use a different View for each data field.
- It also permits each View object to have its own coordinate system for drawing. Views are positioned within the coordinates of their superviews, so when a View object is moved or its coordinate system is transformed, all its subviews are moved and transformed with it. Since a View draws within its own coordinate system, its drawing instructions can remain constant no matter where it or its superview moves on the screen.

The view hierarchy is not the same as the inheritance hierarchy. The inheritance hierarchy is an arrangement of classes; the view hierarchy is an arrangement of objects. They don't parallel each other. You could create an instance of the Box class that had another instance of Box as its superview, for example, and several ScrollViews and Controls as subviews.

The inheritance hierarchy is fixed at compile time. The view hierarchy is dynamic; it can be rearranged as the program runs. A View can be moved from window to window and be installed as a subview first of one superview then of another.

Three instance variables locate a View object within its view hierarchy:

window	The Window object where the View will appear
superview	The object that's immediately above the View in the hierarchy
subviews	A list of all the objects that are immediately below the View in the hierarchy

A View's superview and all the Views above the superview in the view hierarchy are sometimes referred to as the View's *ancestors*. A View's subviews and all the Views below its subviews are known as its *descendants*.

The Frame Rectangle

The location and dimensions of a View object are provided by the instance variable **frame**. **frame** is an NXRect, a structure that defines the essential features of a rectangle, its coordinates and size:

```
typedef float    NXCoord;

typedef struct _NXPoint {
    NXCoord      x, y;
} NXPoint;

typedef struct _NXSize {
    NXCoord      width, height;
} NXSize;

typedef struct _NXRect {
    NXPoint      origin;
    NXSize       size;
} NXRect;
```

This structure was discussed in Chapter 4, “Drawing.”

The **width** and **height** variables specify the dimensions of the rectangle within the superview’s coordinate system; they cannot be negative. The **x** and **y** variables locate one corner of the rectangle, also within the superview’s coordinates. The width and height are measured from this corner along the positive x- and y-axes, so the corner is the one with the smallest x and y values in the superview’s coordinate system.

If the superview’s coordinate system has the positive x-axis extending rightward and the positive y-axis extending upward in the usual manner, the **x** and **y** variables specify the lower left corner. If the superview’s coordinates have been rotated or flipped, the corner may not be at the lower left visually.

NXCoord, NXPoint, and NXSize are defined in the NeXT header file **dpsclient/event.h**. The NXRect structure itself is defined in **appkit/graphics.h**; it’s used throughout the Application Kit to specify the size and location of rectangles.

A View object doesn’t have to draw a rectangle—it could be a circle, for example, a stick figure, or a line of text—but it draws only inside the area specified by its **frame** instance variable. This area is the View’s *frame rectangle*. A default clipping path is constructed around the edge of the rectangle before the object is displayed. You can specify a more restricted clipping path for the View, but you can’t extend it to include any area falling outside its frame rectangle.

The frame rectangle, therefore, defines the boundaries of the View, the tablet on which it can draw. A View object can be thought of as simply a rectangular opening into the window. **frame** specifies the location and dimensions of this opening.

Figure 6-4 shows the frame rectangles of three hierarchically arranged Views.

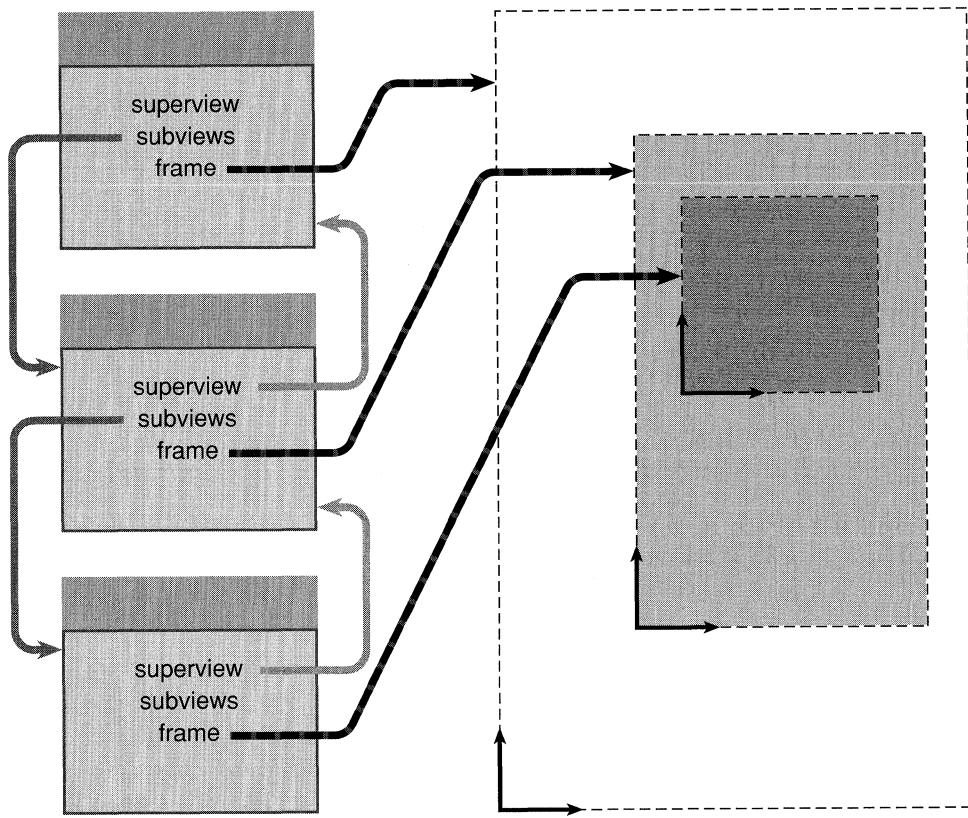


Figure 6-4. Frame Rectangles in the View Hierarchy

Each subview in Figure 6-4 has a frame rectangle that lies totally within the frame rectangle of its superview. This generally is the case. A View's frame rectangle can be larger than its superview's or extend outside it, but only drawing that's within the View's frame rectangle and within the frame rectangles of all its ancestor Views will be visible on-screen.

Figure 6-5 shows three Views similar to those illustrated in Figure 6-4. In Figure 6-5, however, the middle View lies partially outside its superview's frame rectangle. Although the lowest View lies entirely inside its superview, it also lies partially outside an ancestor View, so only the colored portion of it will be visible.

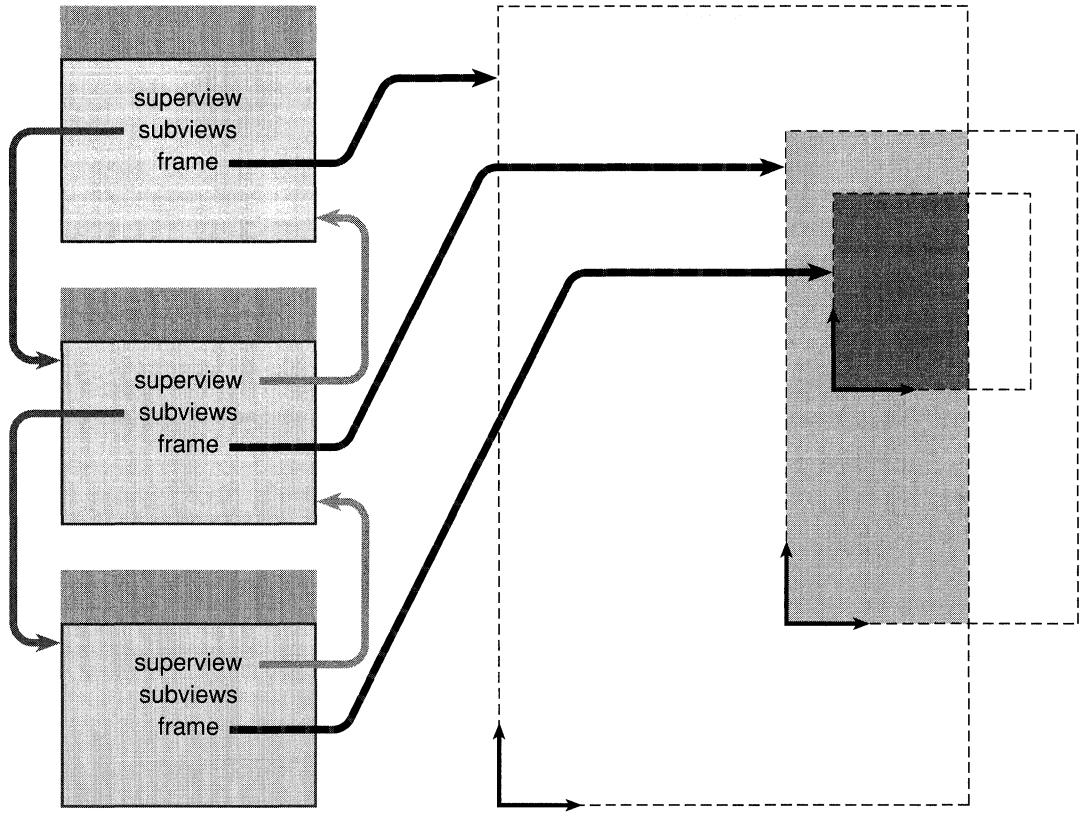


Figure 6-5. Displaced Frame Rectangles

Sometimes, a View contains more material than there's room within a window to display—a View containing the contents of a long document, for example. Such a View can be made the subview of another, much smaller View so that only part of it is visible. With the aid of scrollers, the user can control the larger View's placement within its superview. As the subview moves, different portions of it are scrolled into view. This is illustrated in Figure 6-6. The top View in this illustration would be an instance of the ScrollView class; the middle View would be a ClipView. The largest View, but the one at the bottom of the hierarchy, could be any type of View that you want the user to be able to scroll.

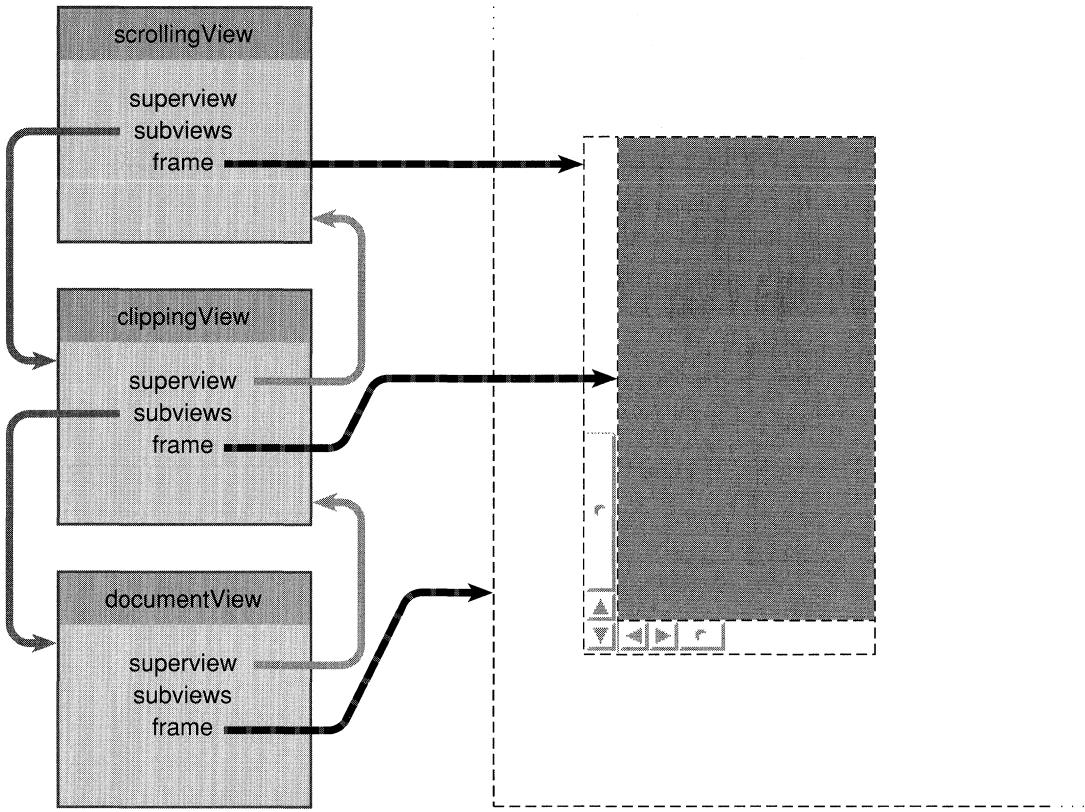


Figure 6-6. Scrolling a View

The frame rectangle is first set by an argument to the class method that creates the View. It can then be modified by methods defined in the View class. These methods are described under “Drawing in the View Hierarchy” in the next chapter.

The entire View—that is, the entire frame rectangle—can be rotated around its origin. Rotation doesn’t affect the shape of the View; it remains a rectangle even though it has been turned and the sides of its frame are no longer parallel to its superview’s x- and y-axes.

Rotation also doesn’t affect the values of the `frame` instance variable; the size of the rectangle remains the same and the corner specified by `frame.origin` stays at the same point no matter what the orientation of the rectangle’s sides.

Figure 6-7 illustrates the same three Views that were shown in Figure 6-4, above. Here, however, the View in the center of the hierarchy has been rotated. Note that its subview rotates with it.

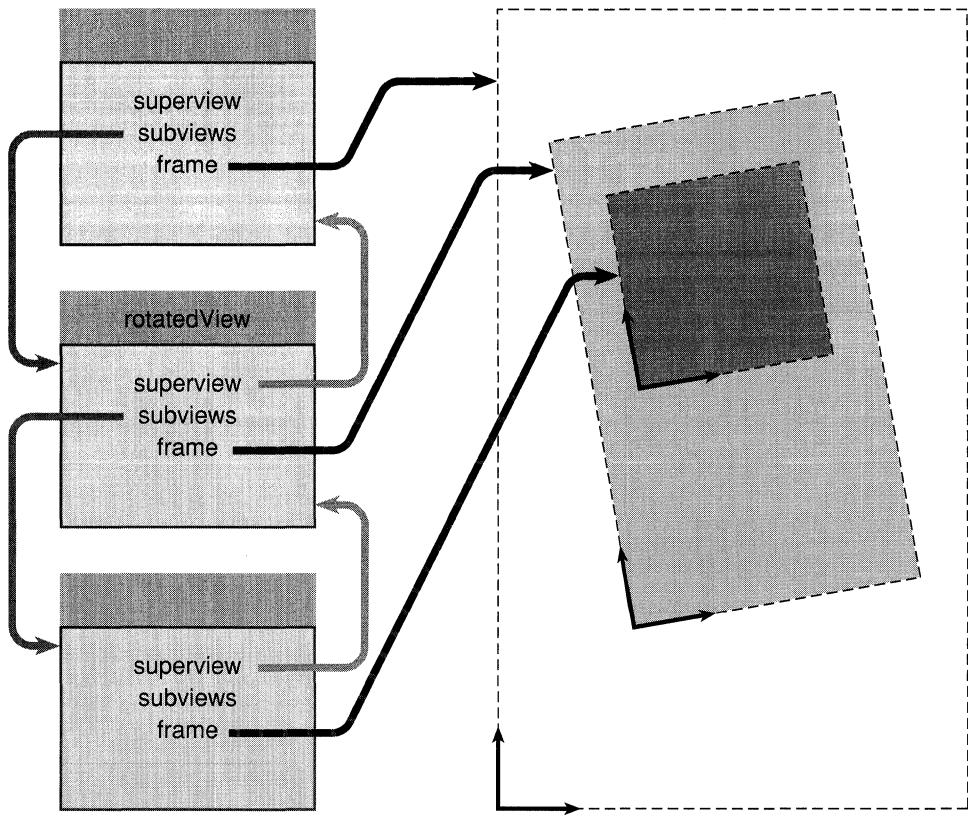


Figure 6-7. View Frame Rotation

Although rotating a frame rectangle is possible, it's not encouraged. It's usually better to rotate the drawing within a View, but to leave the View itself aligned with the screen coordinate system.

Drawing Coordinates

Each View has its own coordinate system for drawing. Before a View draws, its coordinate system is made the current coordinate system for the application.

As a default, a View's coordinate system is the same as its superview's, except that:

- The point recorded in a View's **frame** instance variable becomes the origin (0.0, 0.0) of its drawing coordinates.
- If a View is rotated, its coordinate system is rotated with it; the x- and y-axes stay parallel to the sides of the View's frame rectangle.

Figure 6-8 illustrates the relationship between a View's default coordinate system and its superview's coordinates. The `zooView` object in this figure is located at (350.0, 150.0) in its superview's coordinate system. For the drawing that `zooView` does, this same point is treated as the coordinate origin (0.0, 0.0). In Figure 6-8, `zooView` begins the tip of the crocodile's tail at (50.0, 200.0) as measured from this origin.

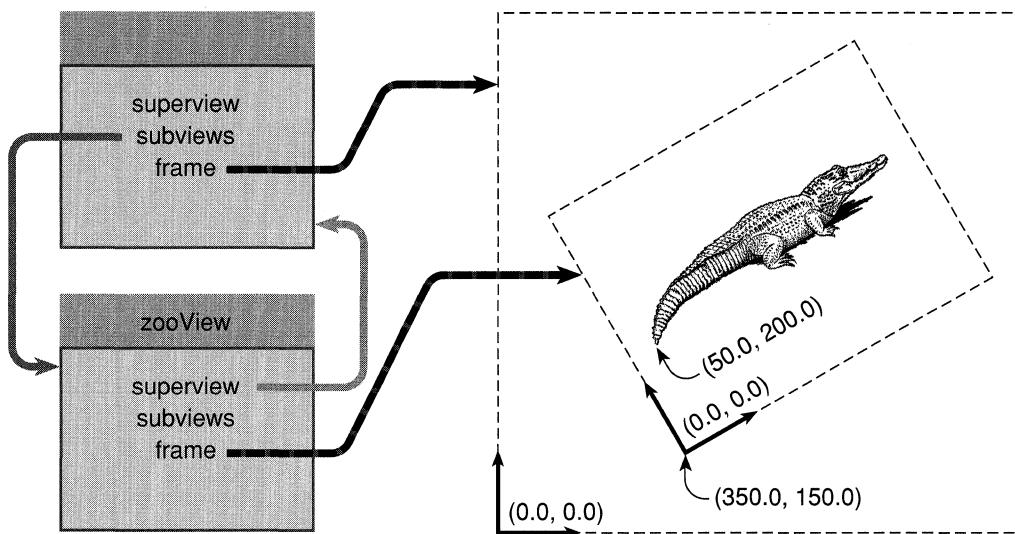


Figure 6-8. Default Coordinates

In Figure 6-8, `zooView`'s frame was rotated about 36° counterclockwise. The default x- and y-axes rotate with the frame, so the picture that `zooView` draws is aligned with the sides of the frame rectangle. The rotation doesn't affect the values in `zooView`'s `frame` instance variable or change the origin of its drawing coordinates.

Note: Frame rotation is shown here only to make the example complete. In practice, although you might rotate the coordinate system temporarily as you draw, you'd rarely want to rotate the entire frame rectangle.

A View can also adopt a different coordinate system by modifying the system it gets by default. Modifications are made with methods that parallel the standard PostScript transformations:

- The coordinate origin can be translated to a point other than `frame.origin`.
- The size of a unit along either the x- or y-axis can be scaled to a dimension other than the one used by the superview.
- The coordinate axes can be rotated around the origin so that they're no longer aligned with the sides of the frame rectangle.

These modifications alter the reference coordinate system that the View uses for drawing, and may affect the appearance of what's drawn, but they don't change the area where the drawing appears—that is, they don't affect the View's frame rectangle.

Besides the default coordinate system and transformations of it, a View can opt for either of two alternative coordinate systems:

- It can keep its superview's coordinate system.
- It can flip its coordinate system so that the origin is in the upper left corner and the positive y-axis extends downward. Flipped coordinates are especially useful for Views that draw text in the normal manner for English (starting at the upper left of the page and proceeding to the lower right).

“Drawing in the View Hierarchy,” in the next chapter, has more detailed information on these options, the methods that can be used to transform the default coordinate system for drawing, and other aspects of View coordinates and displays.

The Bounds Rectangle

The **frame** instance variable locates and sizes a View within its superview's coordinate system, but is of little use when the View comes to draw within its own coordinate system. To draw efficiently, a View often needs to know where it's located and how large it is within its own coordinates; that is, it needs to have its frame rectangle translated into its own coordinate system. The **bounds** instance variable provides this information.

A View's *bounds rectangle* is guaranteed to be the smallest rectangle that completely encloses the View and is expressed in the View's own drawing coordinates. It will usually specify exactly the same area on-screen as the frame rectangle, though stated in a different coordinate system. If a View uses the default coordinate system, **bounds** will be identical to **frame**, except that **bounds.origin** will be (0.0, 0.0).

Views typically use the bounds rectangle to ensure that they don't uselessly attempt to draw images that will never be rendered on the screen. By checking whether the images they wish to draw lie within their bounds, they can avoid sending the Window Server drawing instructions for images that lie outside the View and will consequently be clipped.

Although programs read the **bounds** instance variable, they never set it. **bounds** is automatically updated by methods that change the frame rectangle and by methods that alter the View's coordinate system. These methods are described under “Drawing in the View Hierarchy” in the next chapter.

The Visible Rectangle

The drawing that a View does is clipped not only to its own frame rectangle, but also to every ancestor View above it in the view hierarchy. Therefore, the bounds rectangle by itself is a reliable guide only for Views that aren't scrolled and stay entirely within the frame rectangles of all their ancestors. For Views that fail this test, the View class provides a more reliable indicator of where to draw than **bounds** alone—the *visible rectangle*.

The visible rectangle is the smallest rectangle guaranteed to cover the visible portion of a View's frame rectangle, but stated—like the bounds rectangle—in the View's own reference coordinate system. If a View lies completely inside all its ancestors, the visible rectangle will be identical to the bounds rectangle. If not, the visible rectangle will cover only a part of the bounds rectangle.

Visibility is reckoned only in terms of the view hierarchy; a View may be “visible” even if it's in an off-screen window, or in an on-screen window obscured by other windows.

In Figure 6-5, “Displaced Frame Rectangles,” above, the visible rectangles for the two smaller Views are shaded. In Figure 6-6, “Scrolling a View,” the visible rectangle for the **documentView** is the portion of it that shows through the **clippingView**.

Views don't cache the visible rectangle; it's calculated when needed from the bounds rectangles of the View and from the frame rectangles of all the Views above it in the view hierarchy.

Displaying a View

Like Responder, View acts mainly as an abstract superclass; you'd generally create instances of its subclasses, not of the View class itself. You certainly wouldn't create an instance of View for an object you'd want to see displayed; the View class provides the general mechanism, methods, and instance variables for displaying an object on the screen, but its instances lack methods that can actually do the drawing.

To the general structure provided by View, a subclass must add a **drawSelf::** method with specific drawing instructions in the PostScript language. When a View object receives a display message like this,

```
[aView display];
```

it's brought into focus—its coordinate system is made the application's current coordinate system—and it receives a message to perform its **drawSelf::** method. The display method repeats these steps recursively for each of the View's subviews, so all the Views below **aView** in the view hierarchy are displayed. Views always draw in their own coordinate systems, and subviews are always displayed after (that is, on top of) their superviews.

Figure 6-9 below shows the order in which Views draw. When View A receives a display message, it draws itself and passes the message on to its subviews, B, E, and F. View B is the first of the three to draw, and its subviews, C and D, draw after it and before E. Each branch of the view hierarchy completes drawing before the next branch begins. In this diagram, Views draw in alphabetical order.

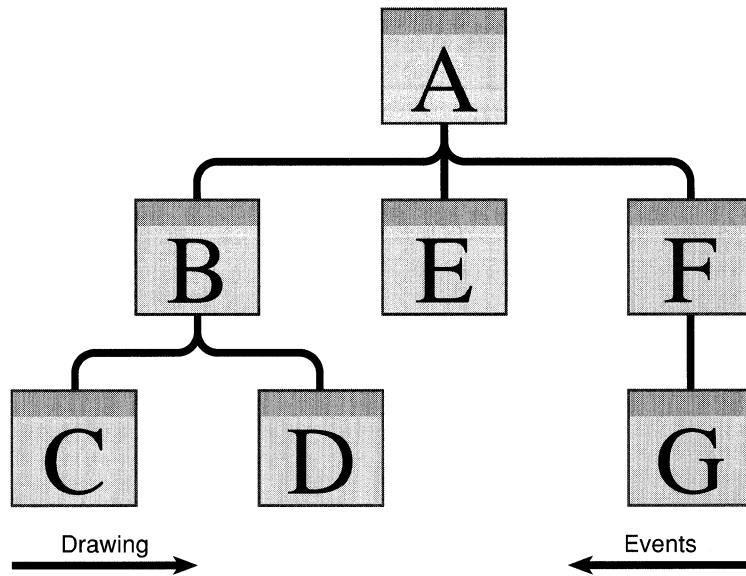


Figure 6-9. A View Hierarchy

See “Drawing in the View Hierarchy,” in the next chapter, for more information on the mechanism for displaying Views.

Associating Events with Views

Users choose a View on-screen by pointing to it (positioning the cursor over its frame rectangle) and pressing the mouse button. This action results in a mouse-down event that records the window where the cursor is located and the cursor’s coordinates. The Application Kit uses this information to find the View and ask it to respond to the event (and to any mouse-dragged and mouse-up events that follow the mouse-down).

Mouse events are associated with Views by working down the view hierarchy, and from the last subview to draw to the first. If, for example, the user presses the mouse button while the cursor is inside View A of the hierarchy shown in Figure 6-9 above, the Application Kit would check to see whether it’s also inside one of A’s subviews, beginning with subview F. If the cursor isn’t inside F, the Kit checks subview E, then B. If it is inside F, the Kit checks F’s subview, G. The result is that the last View to draw, the one on top, is the first to get a chance to respond to the mouse-down event.

Keyboard events are associated with a View in the key window that the user selects with a mouse-down event. If the selected View can display user-editable text or graphics, it becomes the Window's *first responder*. The first responder is given the first opportunity to handle keyboard events and menu commands like Copy, Paste, and Select All that participate in the target-selection paradigm. Every Window can have a first responder, but only the key window's first responder gets keyboard events.

View Subclasses

The objects defined by View subclasses fall into three major groups:

- Views that display data and enable the user to change and manipulate it. The Text object is such a View; it displays user-editable text.
- Views that capture instructions from the user and pass them on to other objects. The subclasses of Control—Button, Slider, Scroller, TextField, Matrix, and Form—define such objects.
- Views that are used in conjunction with other Views, either to enhance or to regulate the display. A Box frames the drawing done by other Views. The frame rectangle of a ClipView defines the area in which a larger subview can be displayed and scrolled; ScrollView adds scrollers to control the display within a ClipView.

Views that belong to the first group are the ultimate consumers of user events; they display the results of the user's actions.

Views that belong to the second group act as intermediaries for actions that ultimately are intended to affect other objects. Because of their importance to program structure, these Views are discussed in a little more detail in the section on the Control class below.

Control

Responder and its immediate subclasses (View, Window, and Application) define the principal event-handling structure for an application. But events alone don't give an application much information about the user's intentions. Because events are closely tied to the computer hardware (keyboard and mouse), there can be but a handful of different event types. Applications therefore need to provide software objects that users can manipulate to give more precise instructions to the application. The Control class is an abstract superclass for objects that play this role. Controls are Views that receive mouse and keyboard events and translate them into application-specific messages for other objects.

A Control's job is to interpret the user's mouse and keyboard actions and ask another object to respond to them. Graphically, Controls provide the user with a display that can be manipulated with the mouse and keyboard—a button that can be pressed, a slider with a

knob that can be dragged, or a text field where data can be entered. Functionally, they convert the mouse and keyboard events they receive into *action messages* for other objects. The object that receives an action message from a Control is its *target*.

Both the action message and the target can be set by the application, so it's possible to adapt instances of a single Control subclass to many uses. One Button might translate the mouse events it receives into a **start:** message; another Button might translate its mouse events into **repeat:** or **stop:** messages. There's a limited set of events, but a virtually unlimited number of control actions.

Since a Menu makes use of a Control object (a Matrix) to contain its list of commands, the message the Menu sends when the user chooses a command is also an action message. Each command has its own action message and can have its own target.

Action messages take a single argument, the **id** of the Control that sends the message. If a target needs more information to accurately respond to an action message, it can send a message back to the Control requesting the information it needs. For example, if a target receives a **changeFilename:** message from a TextField, it can send a **stringValue** message back to the TextField to find the file name the user typed in.

It's your application's responsibility not only to set each Control's target and action message, but also to make sure that the targets can respond. In some cases, objects defined in the Kit can be made the target of a Control. The Text object, for example, can respond to **copy:, paste:,** and **selectAll:** messages, among others. In most cases, however, you'll need to define the target object and implement the methods that will respond to the action messages it receives.

In either case, the Controls themselves can be taken directly from the Kit. The Kit defines six off-the-shelf Control objects that you can use without subclassing—Button, Slider, TextField, Scroller, Matrix, and Form. You can, of course, also define your own Control subclasses.

Action messages should be viewed as application-specific extensions of events. The role of action messages in event handling is discussed under “Action Messages” in the next chapter.

Cell

A Cell is an object that can draw within a View and handle events that are passed to it from the View. Whereas a View receives display messages and events that are distributed down the view hierarchy, a Cell draws and receives events from a View.

A View can record its internal characteristics—what it looks like, how it should respond to the user's action, and so on—in one or more Cell objects. The View itself keeps track of its external characteristics—its size and where it's located in the view hierarchy, for example—as well as of any properties that need to be coordinated among Cells.

Cells are neither Responders nor Views, so they don't initiate drawing or event handling. They may know how to display and highlight a View, or a portion of a View, but can act only on the View's initiative. They may know how to respond to mouse events, but they must get the initial mouse-down event from a View.

Text and Icons

As defined in the Application Kit, instances of the Cell class know how to do two simple drawing tasks:

- Draw a short segment of text within a prescribed rectangle, and
- Draw a bitmap image, an icon, at a specified location.

If a View needs a text label, it can let a Cell draw it. If it needs the application icon displayed within a panel, a Cell is an efficient way to render it. To set a Cell instance to display an icon, you pass it the name of a Bitmap object. To have the Cell display text, you assign it a character string.

For more elaborate drawing tasks, you may need to define View-specific Cell subclasses. Instead of placing drawing code in a View method, `drawSelf::`, it's isolated in the Cell's `drawSelf:inView:` and `drawInside:inView:` methods. In addition to drawing methods, you can endow a Cell subclass with View-specific event-handling methods.

ActionCells

In the Application Kit, most Controls are built around Cells in just this way. A Cell subclass, ActionCell, has instance variables and methods designed specifically for the Control task of translating user events into action messages. For example, the ActionCell class, rather than Control, declares instance variables to record the selector that will be used in an action message and the target object of the message. An ActionCell simply provides that information to its Control when asked; the Control dispatches the action message in response to the events it receives.

There's an ActionCell subclass corresponding to most of the basic Control types, as shown in the chart below:

Control Subclass	ActionCell Subclass
Button	ButtonCell
Form	FormCell
Slider	SliderCell
TextField	TextFieldCell

Some Controls display a single Cell—a Button contains one ButtonCell, a Slider one SliderCell, and a TextField one TextFieldCell.

Other Controls display and manage a group of Cells. A Form can display any number of FormCells. A Matrix can be filled with ButtonCells, SliderCells, TextFieldCells, or any other type of Cell. A Matrix of ButtonCells is, in effect, a coordinated set of buttons, each with its own target and action. A Matrix loaded with MenuCells is used to implement Menus.

Window

Each window in an application is managed by its own Window object. Conversely, every Window object corresponds to a separate window maintained by the Window Server. When you create an instance of the Window class (or of any Window subclass), the Window Server produces the window the instance will manage. The window is identified by a window number, the Window by its object **id**.

To conserve memory and reduce startup time, you can delay creating a window for a Window object until it's placed on-screen; you can also arrange to destroy the window when it's removed from the screen. So it's possible for a Window object to be temporarily without a window. Nevertheless, for the object to fulfill its function, it must at some time be associated with an on-screen or off-screen window.

Note: Throughout this manual, Window objects are referred to with an uppercase "W", and windows created by the Window Server are referred to with a lowercase "w". The distinction should be clear: A window can be on-screen or off-screen, images appear in it, it's affected by PostScript operators like **windowdeviceround**, and so on. A Window has instance variables, it organizes a group of Views, handles events received from the Window Server, and so on. However, since applications affect a window by sending messages to a Window object, uppercase "Window" often covers both the object and the window it manages.

Frame and Content Rectangles

A Window is defined by either of two rectangles, a *frame rectangle* that surrounds the entire window area, including its border, title bar, and resize bar, or a *content rectangle* that encloses just the window's content area. Both rectangles are specified in the screen coordinate system.

The content rectangle is typically used when creating a new Window, and sometimes when resizing it. Values from the frame rectangle are used when relocating a Window.

Like a View, a Window keeps track of its frame rectangle with a **frame** instance variable. But a Window's frame rectangle differs from a View's in some important ways:

- Methods can't change the location or size of a Window simply by altering its frame rectangle. To move or resize a window, you must send instructions to the Window Server. Use the methods described under "Managing Windows" later in this chapter.

- Users can move and resize windows without the application’s intervention. The application is informed of these user actions through messages that they generate. See “Event Messages” in the next chapter for more on these messages.
- Windows can’t be rotated. The sides of their frame rectangles remain aligned with the x- and y-axes of the screen coordinate system.
- Windows don’t draw, so they don’t need a bounds rectangle or a special coordinate system for drawing.

Managing the View Hierarchy

In addition to managing one of the Window Server’s windows, a Window object manages a hierarchy of Views. It has an important part to play both in distributing events among the objects in its view hierarchy and in regulating coordinate transformations among the Views.

Frame and Content Views

Every Window has a view hierarchy with at least two Views: a *frame view* and a *content view*.

The frame view fills the frame rectangle and draws the Window’s border, title bar, and resize bar. It’s at the root of the Window’s view hierarchy; its superview is **nil**. A Window has a frame view even if it has no border or title bar.

The frame view is a private object created by the Window. Applications should refrain from altering it, changing its position in the view hierarchy, or assigning it subviews. Your application won’t draw in the border or title bar, and has no need to handle mouse events outside the content area. For all practical purposes, you can consider the frame view to be *above* the view hierarchy rather than in it.

The content view is the frame view’s only public subview. It fills the entire area of the content rectangle (the area enclosed by the border, title bar, and resize bar). Every Window creates a default content view for itself. It’s not a private object; you can replace it with your own View. The Window records its content view in an instance variable, **contentView**. A method of the same name returns the content view’s **id**:

```
id    curView;
curView = [myWindow contentView];
```

Although the content view has a superview, your application should treat it as if it were at the root of the view hierarchy. For an application, it serves as the principal link connecting the view hierarchy to the Window. To put a View in a Window, install it as a subview of the content view (or as a subview of a View that’s already been installed). You can also replace the default content view with your own.

Coordinate Systems

When a Window is created (or awakened after being unarchived), a Display PostScript graphic state object is created for it. Applications identify the graphics state by a user object, a unique integer:

```
int myGState;
myGState = [myWindow gState];
```

The graphics state object has two important properties for Views that draw within the Window:

- It identifies the window as the output “device” where their drawing will be rendered.
- It records the window’s base coordinate system.

The base coordinate system is the reference point for defining the individual coordinate systems of the Window’s Views. The frame view draws the Window’s border, title bar, and resize bar directly in the base coordinate system. Unless the other Views are assigned graphic state objects of their own, they draw in coordinate systems that are transformations of the base coordinate system.

The value returned by **gState** can be used as an argument to functions like **PScomposite()** and **PSdissolve()**, but you generally don’t need to refer to it. The Application Kit takes care of setting the correct graphics state.

Events

Every Window object keeps an event mask that determines which events the Window Server can associate with the window it manages. Keyboard and mouse events associated with a window are first sent to the Window object. The Window then distributes them to its Views.

The View that the user selects to receive keyboard events is known as the *first responder*. Each Window maintains a **firstResponder** instance variable for the object that should handle the next keyboard event it receives. It constantly updates the instance variable in response to the user’s actions with the mouse; whenever it gets a mouse-down event, it finds the subview where the cursor is located, sends it the event, and tries to make it the **firstResponder** for subsequent events. (See “The First Responder” under “Event Handling” in the next chapter for more information.)

A Window object handles kit-defined subevents—like window-moved and window-exposed—itself; it doesn’t distribute them to its Views.

Window Subclasses

Window has a single subclass, Panel. But the Panel class has several subclasses. Most, such as PrintPanel, FontPanel, and SavePanel, define specific panels that are common to many applications. One is a generic Menu class for all the menus of an application.

Panels are Windows with a special purpose: They hold Views that control other objects associated with other windows of the application. The Views in a Panel are typically Controls of one sort or another; you can think of a Panel as simply a Window container for a cluster of Controls. A Menu has just one Control, a Matrix of MenuCells.

The Panel class gives objects that inherit from it the behavior expected of panels in the user interface:

- A Panel can't become the main window, though by default it can become the key window.
- In general, the only Panels that are visible on-screen are the ones that belong to the application the user is currently working in. By default, the Application Kit hides Panels whenever an application deactivates, unless the Panel is an attention panel.
- When a Panel is closed, it's removed from the screen list but isn't destroyed.
- A Panel passes Command key-down events to the objects in its view hierarchy so that those with keyboard alternatives can respond. See "Keyboard Alternatives" in the next chapter for details.

A Menu is a Panel that displays a single list of commands for the user to choose from. It adds three main features to those defined in the Panel class:

- A Menu can never become the key window (though it can still respond to keyboard alternatives).
- Menus are assigned a tier in the screen list that keeps them in front of all other windows (except attention panels and lists).
- Menus participate in a system of submenus. Any Menu can be made a submenu of another Menu simply by assigning it to a command in the other Menu.

Menu has one subclass, PopUpList, whose instances serve as either pop-up or pull-down lists.

The user's actions on the Controls in a Panel or Menu generate messages for other objects. These messages are discussed under "Action Messages" in the "Event Handling" section of the next chapter.

Application

Every program must have an Application object to supervise its connection to the Window Server, keep track of its Windows, and get its events. It should be the first object created in your program, before any Windows or Views. As one of its first acts, a new Application object connects the application to the Window Server and initializes its run-time environment.

The Application object you create for your program is assigned to a global variable, NXApp. This makes it well-known to the other objects in the application so they can send it messages and avail themselves of its services.

The Application object has four main tasks:

- It receives events from the Window Server and distributes them to other objects.
- It manages all the application's Windows.
- It serves as the default delegate for the application's Listener.
- It keeps global information that's shared by other objects.

The Application object (or its delegate) can also hold application-specific code that defines at least part of what the application is about.

Receiving Events

Soon after an application is launched, NXApp begins getting events from the Window Server and dispatching them to other objects for action. It continues to get events and dispatch them until the application terminates. NXApp is the only Kit object with the ability to receive events from the Server, and the only one with an overview of the whole application.

The section on “Event Handling” in the next chapter has a complete discussion of the Application object’s role in responding to events.

Window Management

The Application object keeps an instance variable, **windowList**, which holds the **ids** of all the Windows associated with the application. The window list lets NXApp:

- Distribute events to the proper Windows.
- Distribute Command key-down events to all on-screen and off-screen Windows (including Menus and Panels) that might have Controls with keyboard alternatives.
- Make sure that drawing appears in the right window.

- Hide and unhide all the application's windows.
- Send messages that update Window displays.
- Find most of the objects in the application, through each Window's view hierarchy and the instance variables of the Views.

Window management tasks are shared by Window objects and NXApp. The methods defined in both classes are discussed together in a later section of this chapter, “Managing Windows.”

Listener’s Delegate

Each application has a Speaker object to send remote messages to other applications and a Listener to receive them. A certain number of the remote messages an application receives come from the Workspace Manager:

- When the user chooses a file in a directory window, the Workspace Manager may send the responsible application an **openFile:ok:** message to have it open the file.
- When the user turns the power off or logs out, the Workspace Manager broadcasts the news by sending a **powerOffIn:andSave:** message to every running application.
- When the user double-clicks a freestanding or docked icon, the application receives an **unhide** message.
- When the user unmounts a disk, applications are informed with an **unmounting:ok:** message.

When the Listener receives these messages from the Workspace Manager, it passes them on to its delegate, which is by default the Application object. The Application object has methods to respond to each of the messages listed above. However, in some cases—such as opening files—it requires help from methods that you need to implement.

Global Services

Because of its central position in a program, the Application object is able to provide other objects with a variety of services. Among other things, NXApp can:

- Activate the application.
- Get events for other objects.
- Terminate the application process.

A principal service is to maintain global information for the application and return it when asked:

- NXApp can say whether the application is the current active application, and whether it's currently hidden:

```
BOOL activeStatus, offScreen;

activeStatus = [NXApp isActive];
offScreen = [NXApp isHidden];
```

- It can supply information about the application's PostScript execution context:

```
DPSContext myContext;
myContext = [NXApp context];
```

- It can return the port where other applications can send it messages:

```
port_t socket;
socket = [NXApp replyPort];
```

- NXApp can identify the Window that holds the freestanding or docked icon that the Workspace Manager created for the application:

```
myIconWindow = [NXApp appIcon];
```

Your application can draw in this Window, but the Workspace Manager owns it.

- It provides the application with a Pasteboard object that it can use to support cut, copy, and paste operations:

```
myPasteboard = [NXApp pasteboard];
```

- NXApp registers and returns the application's main menu:

```
[NXApp setMainMenu:theMenu];
myMenu = [NXApp mainMenu];
```

- It can also register and return global Listener and Speaker objects for the application:

```
[NXApp setAppListener:theListener];
[NXApp setAppSpeaker:theSpeaker];

myListener = [NXApp appListener];
mySpeaker = [NXApp appSpeaker];
```

The Listener and Speaker enable an application to communicate with the Workspace Manager and other applications.

Other services of the Application object are described in later sections of this chapter and in the specification sheet for the Application class in *NeXTstep Reference, Volume 1*.

Application-Specific Code

Because of its central role in an application, NXApp may seem like a good place to put code that's central to your application. You can do this by defining an Application subclass and adding the instance variables and methods you need. An example is shown below.

```
@interface MyApplication : Application
{
    id actor;
}

- setActor:anObject;
- actor;

@end

@implementation MyApplication
- setActor:anObject
{
    actor = anObject;
    return(self);
}

- actor
{
    return(actor);
}

@end
```

This subclass definition does very little. It simply adds the instance variable **actor** and provides the methods **actor** and **setActor:** so that other objects can read and change its value. In all other respects, an instance of the MyApplication class is identical to an instance of the Application class.

Instead of defining an Application subclass, you can centralize your code in another object and make it the Application object's delegate:

```
id myObject;

myObject = [MyClass new];
[NXApp setDelegate:myObject];
```

The delegate will act on behalf of NXApp.

Either way, there are good reasons for putting application-specific code in the Application object (or its delegate). Some of them are listed below.

Overseeing Other Objects

The Application object oversees the application’s Windows. And, as the last few items listed in the section on “Global Services” above illustrate, it maintains program globals for a number of other objects.

As you add objects to your application and program its algorithms, you may be tempted to expand the services of the Application object to include overseeing your objects as well. This can be as simple as adding an instance variable and the methods to set and return its value, as shown above in the example of the MyApplication class.

Action Messages

When an action message from a Control is destined for a target selected by the user, the Application Kit first checks whether the selected receiver can actually respond to the message. If it can’t, the Kit tries to find an alternative receiver. NXApp is the last object given a chance to respond. It’s therefore a good place to put methods that define the application’s default responses to action messages. See “Action Messages” in the next chapter for more on how these messages are dispatched.

Remote Messages

By default, NXApp is the object that’s expected to respond to remote messages from other applications. You can set up your application so that other objects receive remote messages, but the Application object has code to deal, at least in part, with common cases, such as user requests to open another file relayed through the Workspace Manager.

Application-Defined Events

If your application makes use of application-defined events, the method that responds to them should be implemented in the Application object (or its delegate).

Program Framework

A complete application is formed from a variety of objects working together. Each object has its own part (and sometimes several parts) to play. When the user acts and the application receives an event, the Application object generates a message to another object. That object does its part and most likely generates additional messages to still other objects. Those objects may generate their own messages, and so on until the application is ready for another event.

To work together like this, objects must somehow be linked together; each one must know (or be able to find) the appropriate receivers for the messages it sends. Objects usually store this information in instance variables that are initialized when the program starts up and may be altered as it runs. The instance variables define the roles that objects can play within the application; when initialized, they define the framework of the application.

The Application Kit provides a program structure for the applications that use it. It provides, in fact, for three levels of structure, ranging from a core framework of tightly linked objects, to objects in defined roles that are more loosely connected, to ways of freely connecting objects in any configuration you choose.

The Core Framework

Responder and its subclasses define a number of instance variables that point to other objects. They were mentioned in the previous discussion of the principal Application Kit classes and are listed again below for ease of reference.

Class	Instance Variables
Responder	nextResponder
View	superview subviews window
Window	contentView
Application	windowList

When these instance variables are initialized, objects are linked into a working program structure. The framework they define is a network of Responders; they all link Responders to other Responders.

In some cases, an instance variable is initialized when the object it points to is created; in other cases it's initialized when a View is assigned a place in a Window's view hierarchy. Rarely is an instance variable set explicitly; because the connections between the principal event-handling objects of an application are crucial, the Application Kit tries to set them itself as a by-product of other decisions you make.

Figure 6-10 and Figure 6-11 below diagram the connections that are maintained by the core instance variables. Figure 6-10 shows how the Application object, NXApp, is connected to the Windows of the application and how each Window is connected to its view hierarchy.

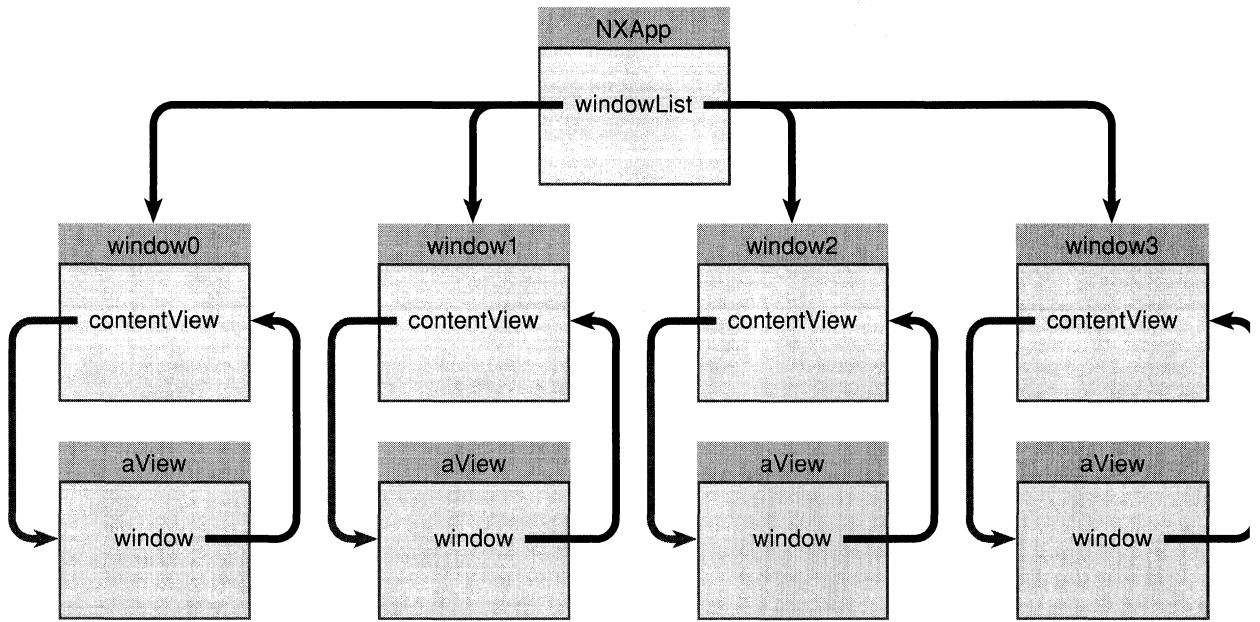


Figure 6-10. Connections to Windows

Figure 6-11 shows how objects are connected within a Window's view hierarchy. The particular Window in this example, **myWindow**, has four Views; **myContents** is the content view and it has two subviews, **frontView** and **backView**. **backView** also has a subview, **longView**.

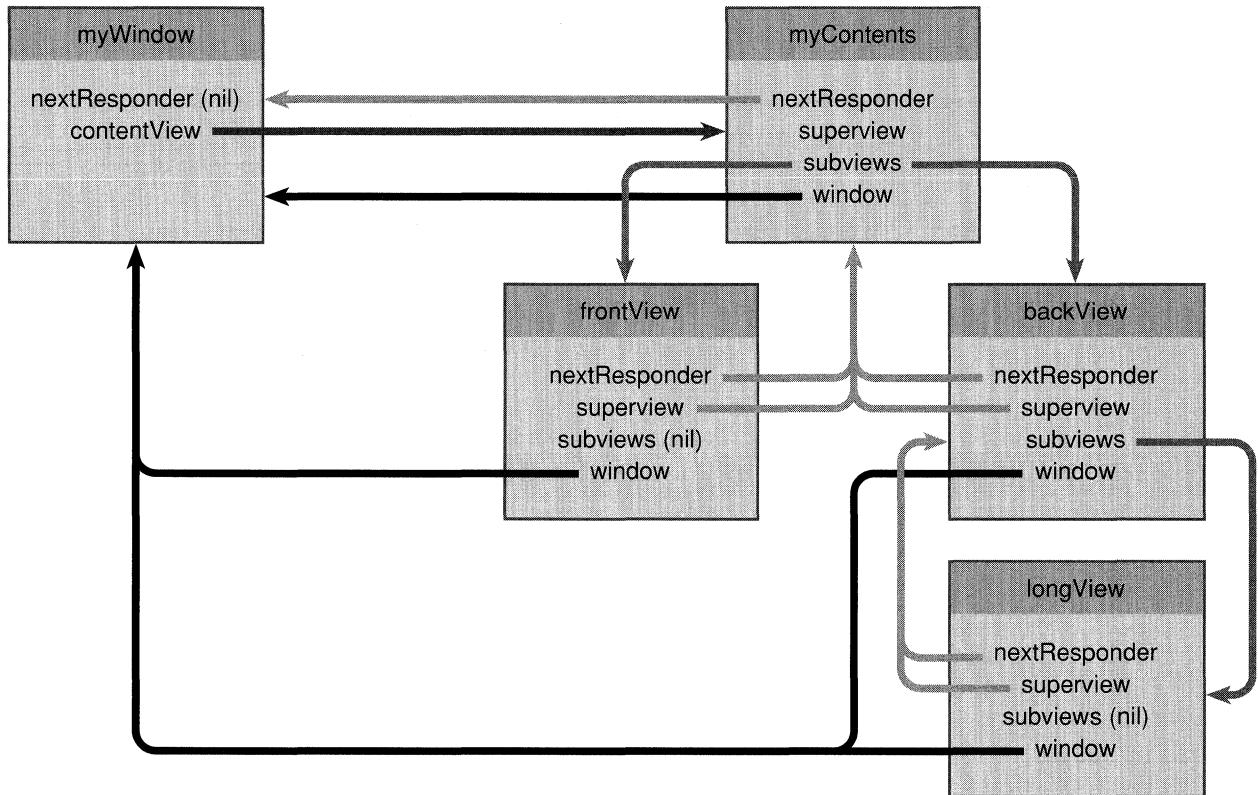


Figure 6-11. Connections in the View Hierarchy

The sections below discuss each instance variable illustrated in Figure 6-10 and Figure 6-11.

windowList

When a new Window object is created, it's added to NXApp's list of Windows. The list lets NXApp keep track of all the Windows in the application and, through the Windows, all the Views in every view hierarchy.

The **windowList** instance variable lists Windows in the order the application creates them (not their order on-screen). It's a List object that's returned by the **windowList** method:

```

id myWindows;
myWindows = [NXApp windowList];

```

Windows (and other objects) keep track of the Application object by its global name, NXApp; they don't need to cache a pointer to it.

contentView

Windows have more than one pointer into their view hierarchies. The most public pointer, and the one that's most important to application programs, is **contentView**. Every Window creates a default **contentView** for itself, since it must have one to function correctly. You can install your own with the **setContentView:** method:

```
id oldView;
oldView = [myWindow setContentView:aView];
```

This method resizes **aView** so that it fits the Window's content rectangle and makes it a subview of the View that draws the Window's border and title bar. It detaches the former **contentView** from the Window's view hierarchy and returns it so that you can free it or reuse it somewhere else.

The Window returns its current content view when asked:

```
id curView;
curView = [myWindow contentView];
```

superview and subviews

The **addSubview:** method links objects into the view hierarchy:

```
[wideView addSubview:myView];
```

This message makes **myView** a subview of **wideView**; both **wideView**'s **subviews** list and **myView**'s **superview** instance variable are adjusted accordingly. If **myView** was already the subview of another View, **addSubview:** would first remove it from the **subviews** list of its former superview before adding it to the end of **wideView**'s **subviews** list.

Because Views are displayed in the order that they appear within the **subviews** list, it's possible for one subview to draw on top of another subview. (Events are associated with subviews in the opposite order, so the View on top will get the event.)

When you assign a View to a superview, you can specify its position in the **subviews** list by using the **addSubview::relativeTo:** method instead of **addSubview:**

```
[wideView addSubview:yourView :NX_BELOW relativeTo:myView];
```

This message puts **yourView** just ahead of **myView** in **wideView**'s list of subviews, where it will draw immediately below (prior to) **myView**. If **yourView** and **myView** overlap, **myView** will be on top. It's also possible, with the **NX_ABOVE** constant, to specify drawing positions above another View.

If the final argument to **addSubview::relativeTo:** is **nil**, **NX_BELOW** places the View in the very first position in the list, and **NX_ABOVE** places it in the very last position.

The **replaceSubview:with:** method removes a View from the **subviews** list and puts another View in its place. Here **myView** replaces **yourView**:

```
[wideView replaceSubview:yourView with:myView];
```

There's no default assignment of **superview**; you must explicitly set it with the **addSubview:**, **addSubview::relativeTo:**, or **replaceSubview:with:** method.

The **removeFromSuperview** method unlinks a View from the view hierarchy; it's deleted from the **subviews** list of its superview and its own **superview** instance variable is set to **nil**, but it isn't freed. Here **yourView** is removed from the view hierarchy:

```
[yourView removeFromSuperview];
```

The **superview** method returns a View's superview, and **subviews** returns its list of subviews, an instance of the **List** class:

```
id parent;
parent = [myView superview];

id children;
children = [myView subviews];
```

The **isDescendantOf:** method returns whether the receiver is on a direct path below another View in the view hierarchy:

```
BOOL isHeir;
isHeir = [myView isDescendantOf:anotherView];
```

You can also test the relationship between two Views by finding their closest common ancestor:

```
id link;
link = [myView findAncestorSharedWith:yourView];
```

This method may return the **id** of either the receiver or the specified View, if one is the ancestor of the other. If the two Views aren't connected to the same Window (the same view hierarchy), it returns **nil**.

window

When a View object is positioned within a view hierarchy, its **window** instance variable is automatically initialized to the Window that owns the hierarchy, as are the **window** instance variables of any descendant Views it may have.

When **myView** is made a subview of **wideView** in the example below (repeated from the section above), its **window** instance variable is updated to match **wideView**'s.

```
[wideView addSubview:myView];
```

The following four methods all have this effect:

```
setContentView:  
addSubview:  
addSubview::relativeTo:  
replaceSubview:with:
```

A **removeFromSuperview** message makes the affected View's **window** instance variable **nil**. The **window** instance variables of all its descendant Views are similarly made **nil**. This also holds true for the Views that are removed from the view hierarchy by **setContentView:** or **replaceSubview:with:** messages.

Every View can identify its Window:

```
id oriel;  
oriel = [myView window];
```

The **window** instance variable gives a View access to information about the drawing environment that's kept by the window. For example, a View can ask the Window for the window number or send the content view a message:

```
int handle;  
handle = [window windowNum];  
  
[[window contentView] display];
```

nextResponder

Every Window, View, and Application object inherits a **nextResponder** instance variable from the Responder class. When a View is assigned a superview, its **nextResponder** is initialized with its superview's **id**. In the **addSubview:** example above, **wideView** was made **myView**'s next responder.

Within a Window, the responder chain mirrors the view hierarchy up to the content view. The **setContentView:** method makes the Window the new content view's next responder. The Window's next responder is **nil**.

The methods that remove a View from the view hierarchy set its **nextResponder** instance variable to **nil**.

It's best to keep the responder chain parallel to the view hierarchy, as **addSubview:**, **addSubview::relativeTo:**, and **replaceSubview:with:** do. But, if you need to, the **setNextResponder:** method lets you override default next responder assignments. Here it adds **lastResponder** to the end of **myWindow**'s responder chain:

```
[myWindow setNextResponder:lastResponder];
```

The **nextResponder** method returns an object's current next responder:

```
next = [myView nextResponder];
```

Outlets

Every application must have an Application object, at least two or three Windows, and some Views to display within the Windows. So every application will have the core framework of Responders discussed above. This framework is self-contained; instance variables like **windowList** and **Superview** store references to other core objects.

Almost every application will need to define its own set of objects to encapsulate behavior that's specific to the application. Some objects will be Responders—Views of a custom design, special types of Panels, an Application subclass. As such, they assume roles within the core framework.

Most applications will also need to define objects that aren't Responders, or define relationships between objects that aren't covered by the core instance variables. An application's network of objects can be extended beyond the core framework through what Interface Builder terms "outlets"—instance variables that point to other objects. The Application Kit provides some outlets that you can initialize, and you can provide others to accommodate the objects you define for your application.

In many ways, outlets are similar to core instance variables like **Superview** and **windowList**; they define receivers for messages that are generated in response to events and remote messages. But outlets mainly differ from the instance variables that define the core framework:

- Outlets are set explicitly, not as a by-product of some other action.
- They can be **nil**; applications usually provide default behavior when an outlet hasn't been initialized.
- An outlet can belong to any class. They can be Responders, but don't have to be. They're often objects that are defined specifically to hold the inner algorithms of the application, as opposed to the code concerned with the user interface. As such, they're usually subclasses of the generic Object class.

The Application Kit defines two outlet instance variables that can extend a connection to objects outside the core:

- Controls send their action messages to a **target**.
- Some objects let a **delegate** take over some of their responsibilities.

The sections that follow discuss the role of these two instance variables in a program's structure.

Delegates

A delegate acts for another object. Instead of defining a subclass to add functionality to a class, the methods and instance variables that would have gone into the subclass are placed in a class definition for an independent object. That object is then assigned to be the delegate of one or more client objects.

There are no default delegates. You must define the delegate's class to give it the functionality your application requires, create an instance of the class, and assign it to a client with the **setDelegate:** method. Here **setDelegate:** makes **myProxy** NXApp's delegate:

```
[NXApp setDelegate:myProxy];
```

delegate returns the receiver's delegate:

```
id theBoss;
theBoss = [NXApp delegate];
```

These Application Kit classes define **setDelegate:** and **delegate** methods:

```
Application
Window
Text
Listener
Speaker
```

The delegate that's assigned to a Window or to the Application object can eliminate the need to define a Window or Application subclass; it can take over most functions you'd be tempted to add to the class. Panels, Menus, and PopUpLists inherit the ability to have a delegate from the Window class.

Although other Responders have delegates, Views (with the exception of Text objects) don't. Applications generally need to define View subclasses to add specific drawing and event-handling behavior to the generic class. The subclass can contain any application-specific code you need to write; a delegate would be extra baggage.

As implemented in the Application Kit, the Text object has enough functionality and options to serve almost all applications; few will need to define a Text subclass. It has a delegate only to give the application control over the user's text input and editing, and over the user's attempts to resize the display.

The delegates defined for Responders (in the Application, Window, and Text classes) extend the event-handling capabilities of those objects. The delegates defined in the Speaker and Listener classes extend the application's ability to send and respond to remote messages. At startup, before processing the first event, the Application object makes sure that a Listener and Speaker are in place and sets itself, NXApp, to be their delegate. The Listener normally entrusts its delegate with the remote messages it receives. But since NXApp can have its own delegate, the Listener defers to NXApp's delegate whenever possible. You can therefore implement the methods that respond to remote messages in an Application subclass or in the class you define for the Application object's delegate.

Delegated Messages

Because a delegate is an object you design, it can do anything you choose for it to do. You could, for example, have it hold the basic algorithms for your application and make it the target for a set of Controls.

In its role as a delegate, it can receive a variety of messages both from the client objects it serves and from other Kit objects that get the delegate's **id** from the client. The Application Kit initiates some messages to delegates as part of the normal process of responding to events and remote messages. Messages are sent only if the delegate implements a method that can respond. In general, the messages sent by Kit objects fall into four categories:

- Messages that notify the delegate of some action the client object took or is about to take.
- Untargeted action messages that the client object could potentially have responded to. Application and Window delegates are the only ones that can receive action messages from Controls. See "Action Messages" under "Event Handling" in the next chapter for a description of how these messages are distributed to delegates (and other objects).
- Remote messages from other applications, as relayed through the Listener. By default, NXApp is expected to respond to remote messages. Its delegate can intervene and take over that function.
- Messages that announce application-defined events and subevents of the system-defined event. It's the Application object's responsibility to respond to these events, and its delegate is given a chance to handle the task. See "Event Handling" in the next chapter for details.

Notification messages are easily recognized by the structure of the methods they ask the delegate to perform:

class { Will | Did } Action : . . .

The method name begins with an indication of the client object's class—"app," "window," or "text." This is followed by "Will" or "Did", depending on whether the message notifies the delegate prior to the client taking action or only after the client has acted. Next is an indication of the action and a colon introducing the method's first argument, the **id** of the object sending the message. A notification message is always sent by the client object itself, so the sender belongs to the class identified by the first word of the method name. Other keywords and arguments may be added at the end.

Prior notifications (those with "Will") generally give the delegate a chance to approve or disapprove of the impending action, or to modify it in some way. For example, a Window sends its delegate **windowWillResize:toSize:** messages as the user drags an outline of the window to a new location. The message gives the delegate a chance to constrain the size of the window.

Notifications after the fact (those with "Did") allow the delegate to coordinate other activities with the actions of its client. For example, a Window delegate receives **windowDidBecomeKey:** messages from a client Window after the Window becomes the key window. The Application object sends its delegate an **appDidInit:** message after the application has been initialized and is ready to receive its first event. This gives the delegate a chance to do any final initialization that may be necessary.

Windows, Text objects, and the Application object can send their delegates notification messages dealing with a wide variety of topics. Each of these messages is described later in this chapter, in Chapter 7, or in Chapter 9 under the appropriate topic. The full set of notification methods are listed below for ease of reference:

Application

appDidInit:
appDidBecomeActive:
appDidResignActive:
appDidHide:
appDidUnhide:
appDidUpdate:

Window

```
windowWillClose:  
windowWillResize:toSize:  
windowWillReturnFieldEditor:toObject:  
windowWillMiniaturize:toMiniwindow:  
windowDidResize:  
windowDidMiniaturize:  
windowDidDeminiaturize:  
windowDidExpose:  
windowDidMove:  
windowDidBecomeKey:  
windowDidResignKey:  
windowDidBecomeMain:  
windowDidResignMain:  
windowDidUpdate:
```

Text

```
textWillResize:  
textWillChange:  
textWillEnd:  
textWillSetSel:toFont:  
textWillConvert:fromFont:toFont:  
textDidResize:oldBounds:invalid:  
textDidChange:  
textDidEnd:endChar:  
textWillStartReadingRichText:  
textWillReadRichText:stream:atPosition:  
textWillFinishReadingRichText:  
textDidRead:paperSize:  
textWillWrite:paperSize:  
textWillWriteRichText:stream:forRun:atPosition:emitDefaultRichText:  
text:isEmpty:
```

The last method in this list lacks a “Will” or “Did.” It’s the exception that proves the rule.

Because a delegate can belong to any class, the Application Kit checks before sending it a message to be sure that it has implemented a method that can respond. You can implement just the methods that you need.

Targets

Targets receive action messages that are generated as the result of user actions on a control. The action message gives application-specific meaning to the events the control receives.

Like a delegate, a target must implement methods to respond to the messages it’s sent. But unlike a delegate, which receives notification messages chosen from a limited and Kit-defined set, a target receives action messages that are set by the programmer.

Because you can set both the target and the method it's to perform, the Application Kit assumes that the target has implemented a method to respond to the action message it's sent. The Kit doesn't check to make sure the target can respond before sending the message.

See "Action Messages" in the next chapter for information on the structure of these messages and the pattern of distribution when no explicit target is set.

Defining an Outlet

The instance variables defined in the Application Kit—**target**, **superview**, **contentView**, and so on—specify certain roles that objects can play in an application.

You can define other roles simply by adding outlet instance variables in a subclass definition. You could, for example, define an object that would act as the delegate for a number of Windows and provide the object with an outlet instance variable for each Window. Or you could define a View subclass with an instance variable that pointed to an object that had a special relationship to the View, in essence a View delegate.

Object-oriented programming makes adding to the program structure in this way fairly easy. Interface Builder makes it even easier. As you build your application, it lets you define outlets for the messages your objects will send. Interface Builder provides each outlet with a method to set its value; you can define a corresponding method to return its current value.

Named Objects

There's an alternative to instance variables as a way of defining the relationships between objects. With Application Kit functions, you can assign an object a name, then later use the name to retrieve its **id** when you want to send it a message. So that more than one object can have the same name, objects are also assigned an owner. The name is a character string; the owner is another object, which can be **nil**.

NXGetNamedObject() returns the **id** when passed a name and owner:

```
id myGame;  
myGame = NXGetNamedObject("gameBoard", NXApp);
```

NXGetNamedObject() returns **nil** if it can't find the object for any reason. A simple misspelling of the name can cause a **nil** return.

Instead of a single name, **NXGetNamedObject()** can also be passed a series of names concatenated together and separated by underscores as illustrated below:

```
id myObject;  
myObject = NXGetNamedObject("aaa_bbbb_cc_ddd", NXApp);
```

The owner (NXApp in the example above) should own the object named first in the list (“aaa” above). The object named first should be the owner of the object identified by the second name. That object should own the object identified by the third name, and so on. **NXGetNamedObject()** returns the object identified by the last name (“ddd” in the example above).

Note: An object name should not include an underscore. If it does, **NXGetNamedObject()** will treat it as a separator dividing one name from another, rather than as part of the name.

You can name an object and specify an owner for it with **NXNameObject()**. This example assigns a name and owner to **self**:

```
NXNameObject ("bartholomewCubbins", self, NXApp);
```

Interface Builder assigns default names to objects, but it permits you to edit or replace the names that are assigned. Interface Builder also assigns owners. Subviews of the Window’s content view are considered to be owned by the Window; other Views are owned by their superviews. The Window’s owner is determined at run time, when the objects are loaded into memory by **loadNibFile:owner:** and **loadNibSection:owner:**. The method’s second argument names the owner of all Windows in the Interface Builder archive file.

NXUnnameObject() breaks the association between a name and an object:

```
NXUnnameObject ("bartholomewCubbins", self);
```

If you pass an object **id** to **NXGetObjectID()**, it returns a name that can be inserted into the series passed to **NXGetNamedObject()**:

```
char *myName;  
myName = NXGetObjectID (self);
```

Managing Windows

Windows are managed through individual Window objects and through the Application object, which keeps a list of all the Windows in the application. This section describes methods defined in both classes.

Setting Up a Window

The principal class method that creates a new Window, **newContent:style:backing:buttonMask:defer:**, labels five arguments. They ask for information about the location and size of the Window, its style and type of buffering, what title bar buttons it should have, and whether the Window Server should defer creating a window for the object until one is needed.

```

id      myWindow;
NXRect  winRect;

NXSetRect(&winRect, 100.0, 200.0, 400.0, 700.0);
myWindow = [Window newContent:&winRect
                      style:NX_TITLEDWINDOWSTYLE
                        backing:NX_BUFFERED
                           buttonMask:(NX_MINIATURIZEBUTTONMASK |
                                         NX_CLOSEBUTTONMASK)
                           defer:NO];

```

The **NXSetRect()** function initializes an NXRect structure. Here it locates **winRect** at (100.0, 200.0) and assigns it a width of 400.0 and a height of 700.0. This rectangle is then used to position the window, as explained in the following table:

Argument	Permitted Values
content	A pointer to an NXRect structure that specifies the size and location of the window's content area in screen coordinates. The border, title bar, and resize bar are drawn around the area specified. All values in the structure are floating-point numbers (NXCoords), but since a window must be aligned on pixel boundaries, the values shouldn't have any fractional parts. If they do, they'll be rounded down to the nearest whole integer.
style	The style of the Window's border and title bar. Window styles are discussed in Chapter 2, "The NeXT User Interface." These constants are provided to specify a Window's style:
	NX_PLAINSTYLE NX_TITLEDSTYLE NX_SIZEBARSTYLE NX_MENUSTYLE NX_MINIWINDOWSTYLE NX_MINIWORLDSTYLE NX_TOKENSTYLE
	In general, Windows created by an application specify one of the first three styles in this list, principally NX_TITLEDSTYLE or NX_SIZEBARSTYLE . The other constants are mainly for the internal use of the Application Kit. (See the discussion of style in Chapter 2.)
backing	The buffering type for the window. The choices are:
	NX_RETAINED NX_NONRETAINED NX_BUFFERED

button mask	The title bar buttons for this window. They can be specified by combinations of these masks: NX_CLOSEBUTTONMASK NX_RESIZEBUTTONMASK NX_MINIATURIZEBUTTONMASK
defer	NX_ALLBUTTONS combines all three masks into a single constant. Although there is no “resize button,” NX_RESIZEBUTTONMASK is what makes the resize bar work. It should always be specified for resizable windows (those with NX_SIZEBARSTYLE). Without it, the resize bar is just a pretty appendage at the bottom of a window. A boolean flag. If it’s YES, the Application Kit won’t ask the Window Server to produce a window for the Window object until the application is ready to place the window on-screen. If the flag is NO, a window is created immediately for the object. All windows are created off-screen and must be placed on-screen with the orderWindow:relativeTo: , makeKeyAndOrderFront: or orderFront: method. These methods are discussed under “Reordering a Window” later in this section. “Deferred and One-Shot Windows” below has more on when it’s appropriate to pass YES as an argument here.

Once the window’s style, buffering type, and title bar buttons have been set, they can’t be changed. There are methods that return the style and button mask:

```
int myStyle, myButtons;

myStyle = [myWindow style];
myButtons = [myWindow buttonMask];
```

The Miniwindow

The Application Kit provides a miniwindow counterpart for each window the user miniaturizes (the miniwindow isn’t created until it’s needed). The Kit takes care of the mechanics of miniaturization through Window’s **miniaturize:** and **deminiaturize:** methods. Messages to perform these methods are generated in response to user actions—clicking the miniaturize button and double-clicking the miniwindow.

The application’s responsibility is limited to determining what image should be displayed within the miniwindow. On the MegaPixel Display, there’s room for a 48-pixel by 48-pixel icon below the miniwindow’s title bar. According to the user-interface guidelines in Chapter 2, if the window holds a document, this should be the same icon as the one the Workspace Manager displays for the document in a directory window. If not, it should be the same as the application icon.

You can set the icon using Window's **setMiniwindowIcon:** method. The following message has the miniwindow counterpart for **myWindow** display the default application icon:

```
[myWindow setMiniwindowIcon:"defaultappicon"];
```

In addition to "defaultappicon," you can pass **setMiniwindowIcon:** the name you've assigned to a Bitmap object or the names of icons stored as sections in the **_ICON** or **_TIFF** segments of the application executable. A Bitmap will be created for the icon when and if it's needed for the miniwindow. (See "Environmental Information" later in this chapter for more on the **_ICON** and **_TIFF** segments.)

The **miniwindowIcon** method returns the name of the icon used in the receiver's miniwindow:

```
const char *mininame;
mininame = [myWindow miniwindowIcon];
```

The Title

If the window has a title bar, the **setTitle:** method can be used to give it a title or modify the current one; **title** returns the current title:

```
[myWindow setTitle:"Product Names"];
aString = [myWindow title];
```

When a title is set, it's displayed immediately, regardless of whether the display mechanism is temporarily disabled. (See the next chapter for information on the display mechanism.)

The Application Kit doesn't provide a default title; a window remains untitled unless you assign one with the **setTitle:** method. The same title is used for both the window and its miniwindow counterpart. Because there's less room in a miniwindow, it may display only the initial part of the title that's set.

Titles that are too long for the space between the title bar buttons are cropped to fit. To be sure that a window is wide enough for its title, use the **minFrameWidth:forStyle:buttonMask:** class method to get the minimum required width:

```
float howWide;
howWide = [Window minFrameWidth:"A Long Window Title"
           forStyle:NX_TITLEDSTYLE
           buttonMask:NX_CLOSEBUTTONMASK];
```

Note that this method returns the width of the frame rectangle, not the content rectangle. Because of the border, the frame rectangle of a titled or resizable window is 2.0 units wider than its content rectangle.

Changing the Close Button

If a window displays an editable document, the application should alter the appearance of the window's close button to reflect whether or not the user has made changes to the document that haven't been saved.

```
[myWindow setDocEdited:YES];
```

The message above has **myWindow** display the close button that indicates unsaved changes, if it isn't already displaying it. The same message with NO as the argument would have **myWindow** display the standard close button.

The **isDocEdited** method returns the last value set by **setDocEdited:**. NO is the default value.

```
BOOL lastSet;  
lastSet = [myWindow isDocEdited];
```

Background Color

A window can be assigned a background color with the **setBackgroundGray:** method:

```
[myWindow setBackgroundGray:NX_WHITE];
```

The default background color is light gray (NX_LTGRAY). **backgroundGray** returns the current background color:

```
float shade;  
shade = [myWindow backgroundGray];
```

Whenever the Window is displayed, its content area is filled with the background color before any of its Views draw, but only if its content view hasn't been registered as an opaque View. Since an opaque content view paints every pixel within the content area, the background color wouldn't show through anyway. (See "Displaying Background Views," in the next chapter, for information on opaque Views.)

Deferred and One-Shot Windows

Each Window object is associated with a window provided by the Window Server. The Window Server usually provides the window when the Window object is created. However, when you create a Window, you have the option of not producing a window for it right away. If you pass YES as the last argument to **newContent:style:backing:buttonMask:defer:**, the Window object will be created but it won't be associated with a window until one of three methods puts it on-screen: **orderFront:**, **makeKeyAndOrderFront:**, or **orderWindow:relativeTo:**. Any of these methods will:

- Make the Window Server produce a window for the Window object.
- Send the Window a **display** message so that all its Views are displayed into its backup buffer.
- Place the window on-screen.

(The **orderFront:** and **orderWindow:relativeTo:** methods are discussed more specifically under “Reordering a Window” below. **makeKeyAndOrderFront:** combines **makeKeyWindow** and **orderFront:** messages into a single method.)

Deferring the creation of a Window object’s window makes sense in two situations:

- If a window might never be used, deferring it saves the memory that would otherwise be allocated to it.
- If the application is large, deferring some of its windows will reduce the time required to launch it.

Once a window is associated with a Window object, the association usually lasts until the Window object is freed. Freeing the object also frees its window. However, you can arrange for the window to be freed whenever it’s taken out of the screen list, by sending the object a **setOneShot:** message:

```
[myWindow setOneShot:YES];
```

A new window will be produced for the Window object each time it’s returned to the screen.

isOneShot returns the value set by the last **setOneShot:** message:

```
BOOL willFree;
willFree = [myWindow isOneShot];
```

Windows that the user might never bring to the screen, or might put on-screen just once, are prime candidates for being deferred and made one-shot. The information panel, rarely used attention panels, and specialized submenus fit this category.

By deferring the creation of the Window object’s window until it’s needed and then freeing it when it’s no longer needed, you can combine the benefits of buffering when the window is on-screen with the benefits of a nonretained window when it’s not.

You should never attempt to draw within a Window that isn’t associated with one of the Window Server’s windows. The display methods discussed under “Drawing in the View Hierarchy” in the next chapter check to be sure that a window exists for a Window object before displaying any of its Views. Dynamic drawing methods are performed only while the window is visible, so there’s little reason to check. However, if you write any independent code that can draw in an off-screen window, you need to be certain that a window exists before attempting to draw. If the **windowNum** method doesn’t return an integer greater than 0, there is no window to draw in.

Hiding Panels

By default, when the active application deactivates, the Application Kit hides its panels. However, you can choose to override this default and leave a panel on-screen even when its application isn't active:

```
[myPanel setHideOnDeactivate:NO];
```

The **doesHideOnDeactivate** returns the value set by the last **setHideOnDeactivate:** message:

```
BOOL willHide;
willHide = [myPanel doesHideOnDeactivate];
```

Chapter 2 gives guidelines for when it's permitted to leave a panel for a deactivated application on-screen. It should be a rare occurrence.

Window Status

Although users generally select the key window and main window, there's a method that lets the application do it:

```
[myWindow makeKeyWindow];
```

This method is most often used to designate the initial key window when the application first starts up. When a standard window is made the key window, it also becomes the main window. (The **makeKeyWindow** method can be performed in conjunction with **orderFront:** by a single message to **makeKeyAndOrderFront:.**)

The **isKeyWindow** and **isMainWindow** methods return the status of the receiving object:

```
BOOL nowKey, nowMain;

nowKey = [myWindow isKeyWindow];
nowMain = [myWindow isMainWindow];
```

The Application object can identify the current key window and main window:

```
id keyWin, mainWin;

keyWin = [NXApp keyWindow];
mainWin = [NXApp mainWindow];
```

These methods return **nil** if the key window or main window isn't in the current application.

For more detailed information on the methods that determine and reflect window status, see “The Key Window and Main Window” in the next chapter.

Physical Management

There are three ways that an application can manipulate the window associated with a Window object. It can:

- Move the window,
- Resize it, and
- Reorder it in (or out of) the screen list.

These operations are accomplished through messages to the Window object, which, in turn, generates instructions for the Window Server.

Moving a Window

The **moveTo::** method repositions a window within the screen coordinate system:

```
[myWindow moveTo:x :y];
```

Its arguments, (x, y) , give the new location of the lower left corner of the Window's frame rectangle in screen coordinates. Figure 6-12, below, illustrates the effect of the **moveTo::** method. Note that when the window moves, its coordinate system and all its contents move with it.

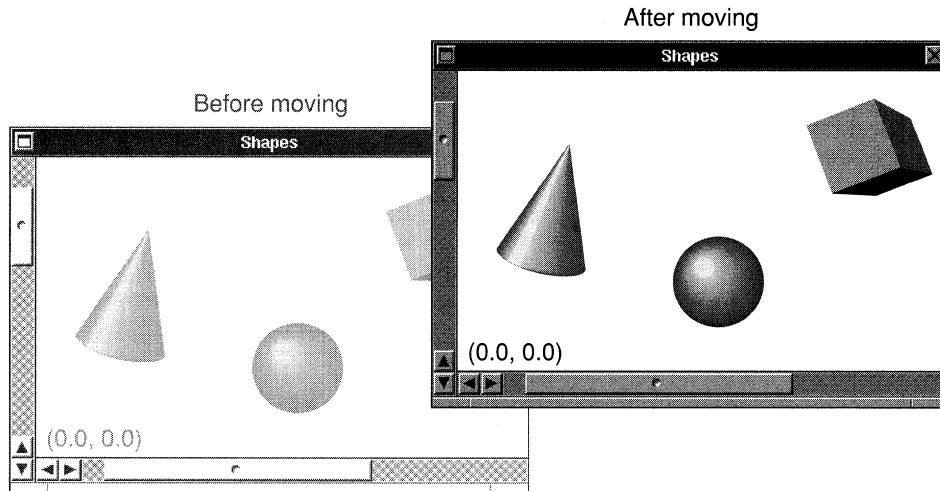


Figure 6-12. **moveTo::**

Another method, **moveTopLeftTo::**, can be used to reposition a window by its top left corner:

```
[myWindow moveTopLeftTo:x :y];
```

moveTopLeftTo:: is the same as **moveTo::**, except that it positions the top left, rather than the lower left, corner of the window. If the height of the window's frame rectangle were subtracted from the y coordinate, a **moveTo::** message could be used with the same effect as **moveTopLeftTo::**.

These two methods move the window from its starting location to its destination in a single step; the window isn't drawn in any intermediate positions. Note that unlike the class methods that create a Window, they position the window's frame rectangle, not its content rectangle.

The **moveTo::** and **moveTopLeftTo::** methods don't generate window-moved subevents (of the kit-defined event). Since the new location of the window is evident from the method's two arguments, none is needed. Window-moved subevents are generated when the user moves a window, not when the application does. The subevent serves as the application's only notice that something has happened.

Typically, users move a window by dragging it by its title bar, but you can also let users drag a window by a point within its content area. The **dragFrom::eventNum:** method specifies the point and requests the Window Server to move the window in response to the user's action. A **dragFrom::eventNum:** message should be sent only after receiving the mouse-down event that initiates dragging. Its first two arguments give the current location of the cursor within the window's base coordinate system, and can be taken from the event record of the mouse-down event. The third argument is the event number of the mouse-down event, also taken from its event record.

```
[myWindow dragFrom:eventPtr->location.x :eventPtr->location.y  
eventNum:eventPtr->data.mouse.eventNum];
```

The Window Server moves the window so that the cursor stays at the same location within the window. When the user releases the mouse button, it generates a window-moved subevent to notify the application. The user's mouse-dragged events are trapped by the Window Server; they're not sent to the application.

The **center** method positions a window so that it's centered in the top two-thirds of the screen. That's where attention panels should come up so that users can't miss or ignore them.

```
[myPanel center];
```

This method adjusts the position of the window, if necessary, to keep its title bar on-screen.

Resizing a Window

The **placeWindow:** method resizes and repositions a window within the screen coordinate system. Its argument is a pointer to an NXRect structure with the new location and dimensions of the window's frame rectangle:

```
NXRect newRect;  
  
NXSetRect (&newRect, 300.0, 500.0, 700.0, 600.0);  
[myWindow placeWindow:&newRect];
```

As illustrated in Figure 6-13 below, the window both moves to a new location and assumes a new shape.

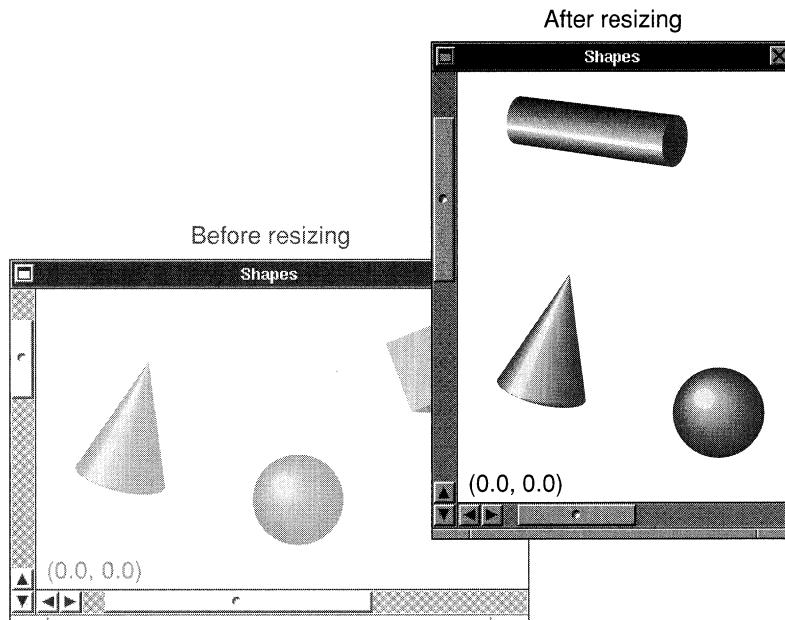


Figure 6-13. **placeWindow:**

However, **placeWindow:** is an expensive way to move a window when compared to **moveTo::** and **moveTopLeftTo::**. The latter two methods rely on the Window Server to shift the window's display and coordinate system to a new location, but **placeWindow:** must reinitialize the window's coordinate system and clipping path and redisplay all the Views within the window after it has moved. For this reason, **placeWindow:** should be used only to resize a window, not to move it.

To make what's going on clear to the user, it's best not to mix resizing with moving. To avoid the appearance of moving a window when you resize it, keep a corner or side of the window constant, or resize the window by expanding it outward (or collapsing it inward) on all sides at once.

Figure 6-13 above disregards this advice. It appears to move the window because, in a sense, its top and right sides are expanded outward while its bottom and left sides are collapsed inward.

sizeWindow::, an alternative to **placeWindow:**, keeps the lower left corner of the window constant while adjusting its width and height.

```
[myWindow sizeWindow:width :height];
```

Its arguments specify the new width and height of the window's content rectangle (not its frame rectangle) in screen coordinates.

Because the new size of the window is apparent from the arguments to **placeWindow:** and **sizeWindow::**, neither method generates a window-resized subevent (of the kit-defined event).

Reordering a Window

The **orderWindow:relativeTo:** method alters the position of a window in the Window Server's screen list. The first argument specifies whether the receiving Window is to be placed above another window in the list, placed below another window, or left entirely off the list. The choices are:

```
NX_ABOVE  
NX_BELOW  
NX_OUT
```

The second argument is a window number that identifies the window the receiving object is to be ordered above or below, or 0 if the receiver is to be ordered above or below all other windows in its tier. If the first argument is NX_OUT, the second argument is ignored. This message takes **myWindow** out of the screen list:

```
[myWindow orderWindow:NX_OUT relativeTo:0];
```

The **orderWindow:relativeTo:** method won't let you order a window into another tier. If the second argument names a window in a higher or lower tier, the receiving Window will be reordered so that it's as close to that window as possible, but it won't be ordered out of its own tier.

It's generally not very useful for an application window to be ordered behind the workspace window (the window that provides the dark gray background on-screen), since users wouldn't be able to find it there. Therefore, the message below puts the receiving Window at the back of its tier but keeps it just in front of the workspace window. You can think of the workspace window as being in a tier of its own.

```
[myWindow orderWindow:NX_BELOW relativeTo:0];
```

The Window class defines three methods as shortcuts for common reordering cases. Each method takes a single argument, an **id**, so that it can be used in action messages. The method itself doesn't make use of the argument.

```
[myWindow orderOut:nil];
[myWindow orderFront:nil];
[myWindow orderBack:nil];
```

The **orderOut:** method removes the receiving Window from the screen list; it has the same effect as the first **orderWindow:relativeTo:** example shown above. The **orderFront:** method puts the receiving Window at the front of its tier in the screen list. **orderBack:** puts a window at the back of its tier, but in front of the workspace window; it's a shorthand for the second **orderWindow:relativeTo:** example shown above.

A fourth method, **makeKeyAndOrderFront:**, combines two operations that often go together—making a Window the key window (as **makeKeyWindow** does) and putting it at the front of its tier (as **orderFront:** does). The user interface requires a window to come forward when it becomes the key window.

```
[myWindow makeKeyAndOrderFront:nil]
```

When the key window (or main window) is taken out of the screen list, it loses its key window (or main window) status. The Application Kit tries to make another window in the active application the new key window (and main window). However, simply reordering windows within the screen list or putting a window that was previously out of the list in, has no affect on any window's status. Moving another window in front of the key window doesn't change the key window.

Getting Information about a Window

NXApp and Window objects provide information about the application's Windows in response to messages sent by other objects.

Many of these messages were illustrated earlier in this chapter. For information about properties that are set when a Window is established, see "Setting Up a Window" above. For information on methods that report a Window's status as the key window or main window, see "Window Status" earlier in this section and "Selecting an Application, Window, and View" in the next chapter. Methods that provide other information are described below.

Frame and Content Rectangles

The **getFrame:** method takes a pointer to a rectangle (**NXRect ***), which it initializes with the location and size of the Window in screen coordinates.

```
NXRect myFrame;
[myWindow getFrame:&myFrame];
```

getFrame: provides the Window's frame rectangle. Given the frame rectangle, the **getContentRect:forFrameRect:style:** class method can find the content rectangle:

```
NXRect myContent;
[Window getContentRect:&myContent
    forFrameRect:&myFrame
    style:NX_TITLEDSTYLE];
```

Given the content rectangle, **getFrameRect:forContentRect:** can find the frame rectangle:

```
[Window getFrameRect:&myFrame
    forContentRect:&myContent
    style:NX_TITLEDSTYLE];
```

Window Numbers

The **windowNum** method returns the receiver's window number:

```
int which;
which = [myWindow windowNum];
```

If the receiving Window object isn't currently associated with a window, this method returns -1.

The Application object can find the **id** of the Window object associated with a window when given its window number:

```
id theWindow;
theWindow = [NXApp findWindow:theNum];
```

Screen List Information

A Window can return whether it's currently in the screen list:

```
BOOL inList;
inList = [myWindow isVisible];
```

The `isVisible` method returns YES even if the window is completely covered by other windows. This method merely tests whether the window is somewhere in the screen list.

The Application object can identify of the frontmost window in the list. It will generally be a menu or an attention panel:

```
id topWin;
topWin = [NXApp frontWindow];
```

NXApp can also provide the ordering of all the application's windows in the screen list:

```
int *listPtr, howMany;
[NXApp getScreenList:&listPtr count:&howMany];
```

getScreenList:count: produces an array that lists all the on-screen windows belonging to the application, ordered as they are in the screen list (the front window first). It returns, by reference, a pointer to the array and the number of windows listed. The array is a list of window numbers, not **ids**. To find the **id** of a Window, pass the window number as the argument in a **findWindow:** message:

```
id theWin;
theWin = [NXApp findWindow:(listPtr+3)];
```

Environmental Information

Applications run in an environment that's determined in part by the hardware, in part by the Mach operating system, the Window Server, and the Application Kit, and in part by choices made by the user. The Kit defines methods and functions that supply the environmental information an application might require:

- The **getScreenSize:** method, defined in the Application class, supplies the size of the screen where the application's windows are displayed:

```
NXSize theScreen;
[NXApp getScreenSize:&theScreen];
```

You should use this method rather than assume a fixed screen size.

- The **hostName** method provides the name of the machine where the application is running:

```
const char *where;
where = [NXApp hostName];
```

- Application’s **appName** method provides the name of the application:

```
const char *me;
me = [NXApp appName];
```

The name is the one assigned while compiling the program, as explained under “Segments” below, or if no name was assigned, the name passed in the **argv** array.

- The name of the current user is returned by the **NXUserName()** function:

```
const char *who;
who = NXUserName();
```

- The user’s home directory is returned by **NXHomeDirectory()**:

```
const char *domicile;
domicile = NXHomeDirectory();
```

When the Workspace Manager launches an application, it sets the application’s current working directory to the user’s home directory. The standard UNIX function **getwd()** returns the current working directory, and **chdir()** changes it.

If you change the current working directory so that it’s on a device—an optical disk—that the user wants to unmount, the Application Kit may change it back to the user’s home directory (or to the root directory if for some reason there is no home directory). The Kit makes the change in response to an **unmounting:ok:** remote message from the Workspace Manager. Since remote messages are received between events, you can be sure that the current working directory will remain constant during the response to an event. But you may need to check before each new event to be sure that it hasn’t been changed “behind your back” between events.

You can avoid behind-the-back changes by implementing an **appUnmounting:** method in the Application object’s delegate. Your method will then need to handle those cases in which the user wants to unmount the disk where the current working directory is located.

Command-Line Arguments

Information entered on the command line when launching a C (or Objective-C) program is passed to the **main()** function as a pair of conventional arguments, **argc** and **argv**. (**argv** points to an array of character strings containing the command-line arguments, and **argc** counts the number of strings in the array.)

To make command-line arguments available from any point within an application, including from within a class definition, the loader writes them to a well-defined location and the Application Kit provides these global variables to refer to them:

```
int      NXArgc;
char **NXArgv;
```

It's therefore unnecessary for the **main()** function to deal with command-line arguments itself.

When the Workspace Manager launches an application, there's no command line where users can specify program options and parameter values. Rather than choose options on the command line, users make choices once the application is launched and activated. An application can record these choices in the defaults database for use the next time the application is launched. It might, for example, remember where the user placed various panels, or which font was preferred.

Nevertheless, the Workspace Manager uses command-line arguments to pass useful information to the applications it launches. If, during testing, you launch your application from a Terminal window, you can also use the command line to pass it test values for various program parameters.

Information passed on the command line as pairs of argument strings in the format

-parameter value

can be read by the functions that prepare a table of default parameter values for an application. The first string begins with a hyphen and names a parameter; the following string assigns the parameter a value. After this information is recorded, the arguments are stripped from the NXArgv array and NXArgc is adjusted accordingly.

Values assigned on the command line in this way override values taken from the user's defaults database. (See "The Defaults System" in Chapter 10 for more information on how command-line arguments contribute to the system of default parameters.)

Arguments that you can pass on the command line when debugging your application include these:

Parameter	Value
NXShowPS	If not NULL, all PostScript code sent to the Window Server, and all values returned from the PostScript interpreter to the application, will be written (in ASCII rather than binary form) to the standard error stream.
NXShowAllWindows	If not NULL, all windows will be placed on-screen, including those that normally stay off-screen—such as windows that store images that are composited to other, on-screen windows.

The arguments passed by the Workspace Manager are read and stripped from NXArgv when the Application class is initialized, just before the **new** message is sent. They define these public parameters:

Parameter	Value
NXOpen	The full pathname of the file the application is to open.
NXOpenTemp	The full pathname of a temporary file the application is to open. Temporary files should be deleted before the application quits.
NXHost	The name of the host machine. The Workspace Manager provides this information only when it's run remotely (on a different machine).
NXAutoLaunch	A value other than 0 if the application was launched by the Workspace Manager at login, and undefined otherwise.

When an application is launched by the Workspace Manager, NXArgv[0] is guaranteed to contain the full pathname of the application's executable file. By stripping the last element from the pathname, an application can obtain the path for the file package where auxiliary files essential to the application are located.

Segments

The Mach object file format, the default format for the NeXT computer, allows you to create segments in the executable file and store whatever information you need there. The software kits take advantage of this facility to keep various kinds of information—archived objects, sound files, icons, and other images—in close association with the code that requires it.

Some segments are created by the compiler and can't be altered or added to:

<u>TEXT</u>	The text of the program's executable instructions
<u>DATA</u>	Initialized and uninitialized data
<u>OBJC</u>	Data structures required by the run-time system.

Other segments can be created with the **-segcreate** option to the link editor:

```
cc ... -segcreate ANIMATION paneOne image1.tiff
```

This example creates the ANIMATION segment and copies the **image1.tiff** file into it as a section named “paneOne.” If the ANIMATION segment had already been created, the **-segcreate** directive would simply have added another section to it. Usually sections keep the same name as the file.

Some segments have well-established names:

__NIB	Archived objects that were created using Interface Builder.
__SND	Sound files required by the program.
__ICON	Files in Tag Image File Format (TIFF) that specify the icons that the Workspace Manager should display for the application and its files.
__TIFF	Other TIFF images used in the program.

You can add sections to these segments as required by your application. The information you store there can be read by methods and functions defined in the software kits. For example, the **findBitmapFor:** method will create a Bitmap object from a section of the __TIFF or __ICON segment.

The first section of the __ICON segment isn't an icon; it's a header, appropriately named “__header.” It specifies how the Workspace Manager is to use the icons stored in the segment. Each line in the header describes a different icon—a different section within the segment. A line contains four pieces of information, separated by tabs:

1. The letter “F” (or “f”) if the icon should be displayed for just one file, or the letter “S” (or “s”) if the icon should be displayed for all files bearing a certain extension (or “suffix”).
2. The name of the file, or the characters in the extension (without the period).
3. The name of the application. This is the name that the Application object’s **appName** method will return.
4. The name of the section. The section containing the application icon—the one that will be displayed for the application executable—should be named “app” (and no section should be named “__tiff” or “__TIFF”).

The following is an example of a header section:

F	Crossword	Crossword	app
S	pzl	Crossword	puzzle.tiff
S	pdrome	Crossword	palindrome.tiff
S	agm	Crossword	anagram.tiff

This header corresponds to linker instructions such as these:

```
-segcreate __ICON __header Crossword.iconheader
-segcreate __ICON app crossword.tiff
-segcreate __ICON puzzle.tiff puzzle.tiff
-segcreate __ICON palindrome.tiff palindrome.tiff
-segcreate __ICON anagram.tiff anagram.tiff
```

Note: Segment names are usually all uppercase letters; section names are lowercase or mixed case. The double underscore prefixes are used only to prevent potential clashes with other names.

Application Kit Conventions

The Application Kit imposes a small number of constraints on the Objective-C code you write. None of them constrain what you can do, only how you can go about doing it.

Reading and Writing Instance Variables

Because an object's instance variables are private to the object, Application Kit classes provide methods that can be used to modify them and read their values. For example, the **setFloatValue:** method assigns a value to a SliderCell object, and **floatValue** returns its current value as a floating-point number:

```
float myVal;  
  
myVal = [mySliderCell floatValue];  
[mySliderCell setFloatValue:212.0];
```

An object has direct access to all its instance variables, so it can read them without sending a message. A SliderCell subclass could look directly at its **value** instance variable:

```
if ( value >= 0.0 )  
    [self doSomething];
```

Although they can be read directly, inherited instance variables should be set only through the methods provided. This is because these methods often do more than simply set the instance variable; they have side effects that the Application Kit counts on. For example, the **setFloatValue:** method makes sure **value** stays within maximum and minimum bounds. Similarly, the methods that change a View object's **frame** and **superview** instance variables automatically update other instance variables. To function accurately, the Application Kit requires that these correspondences be maintained.

Writing Methods That Create Instances

In the Application Kit, class methods that create a new instance generally begin with the word "new." This is usually followed by one or more labeled arguments. For example, three different methods can create an instance of the Button class:

```

myButton = [Button new];
yourButton = [Button newFrame:&rect];
ourButton = [Button newFrame:&rect
                           title:@"Panic"
                           tag:-1
                           target:anObject
                           action:@selector(relax:])
                           key:'p'
                           enabled:YES];

```

The arguments to these methods serve to initialize the new object (they're explained in Chapter 9). The more arguments a method has, the more freedom it gives you to determine the character of the object at the outset.

Regardless of the number of arguments, each method is guaranteed to return a working, fully initialized object, even if some of its instance variables are initialized to 0. To make good on this guarantee, two things are required:

- Each method must be linked, directly or indirectly, to the **new** method defined in the Object class. Object's **new** method is the one that allocates memory for the new object's instance variables.
- Each method must initialize the new object. Methods with more arguments let you do much of the initialization yourself; methods with fewer arguments must supply default values of their own.

In the Application Kit, methods with fewer arguments work by passing default values to the method with the greatest number of arguments. For its part, the method with the greatest number of arguments maintains the class link to Object's **new** method. Through a message to **super**, it performs a class method defined somewhere farther up the inheritance hierarchy; that class method sends its own message to **super** to perform a method it inherits, and so on. This chain of messages eventually leads back to the **new** method in the Object class.

Suppose, for example, that a Kit class defines three instance-creating methods, **new**, **newArg:**, and **newArg:arg:**. The **new** method would have the receiving class object perform the **newArg:arg:** method:

```

+ new
{
    return( [self newArg:0 arg:@"default"] );
}

```

The **newArg:** method would do the same:

```

+ newArg:(int)anInt
{
    return( [self newArg:anInt arg:@"default"] );
}

```

The **newArg:arg:** method, the one with the most arguments, would perform an inherited method:

```
+ newArg:(int)anInt arg:(char *)aString
{
    self = [super newArg:anInt];
    [self setMyString:aString];
    return self;
}
```

These conventions matter when you create an instance of a class or define a subclass. The method with the most arguments is the most efficient and direct way to create an instance; the other methods can lead to initializing and reinitializing the object, first with default values, then with values more appropriate to your application.

Defining a Subclass

When you define a subclass of an Application Kit class, it isn't absolutely required that you write your own methods to create instances for it; inherited class methods will create objects belonging to a subclass as well as to the class where they're defined. But you may need your own class methods to initialize objects of the subclass differently so that their instance variables have reasonable values for your program.

If you do define your own instance-creating methods for a Kit subclass, it's important that they follow the Kit conventions. Suppose, for example, that you decide to override the **newArg:arg:** method illustrated above. If you have your version perform the version of **new** it inherits,

```
+ newArg:(int)anInt arg:(char *)aString
{
    self = [super new];
    ...
}
```

your method will perform the **new** method and the **new** method will perform yours, in a never-ending loop. In its message to **super**, a subclass method should always perform the superclass method with the most arguments.

The convention is simply this: To ensure that Object's **new** method is performed, one instance-creating method in each class is linked to an instance-creating method defined in a class farther up the inheritance hierarchy. Within each class, it's the method with the greatest number of arguments that maintains this link.

Covering Inherited Methods

When you define a subclass, you should guarantee that all the instance-creating methods it inherits produce reasonable objects for the subclass. This may mean overriding the inherited methods to cover them with your own versions.

Two of the example methods illustrated above cover inherited methods: **new** covers the **new** method defined in the Object class, and **newArg:** covers the **newArg:** method used in **newArg:arg:**'s message to **super**.

Covering inherited instance-creating methods makes the class you define more portable to other applications. If you leave an inherited method uncovered, someone else may use it to produce incorrectly initialized instances of your class.

Returning self

As illustrated in the preceding section, class methods that create a new instance return the instances they create.

Instance methods, like C functions, can return a variety of different values—**ints**, **floats**, pointers to structures, and so on. Those that don't have meaningful return values of their own return **self** (the object that receives the message). This permits messages to be chained together, avoiding the need for temporary variables for the receiver. This single line of code,

```
[[[ButtonCell newTextCell] setTitle:@"Keep"] setType:NX_SWITCH];
```

can substitute for these four:

```
id theCell;  
  
theCell = [ButtonCell newTextCell];  
[theCell setTitle:@"Keep"];  
[theCell setType:NX_SWITCH];
```

Tags

Since object **ids** are determined only at run time, it's sometimes convenient to assign objects integer tags that can be used to identify them at compile time. Views and Cells are prime candidates for tag identifiers, since an application is likely to have many of them. In the Application Kit, the Control and Text subclasses of View define a **tag** instance variable for their instances, as does the ActionCell class.

The **tag** method returns the receiver's tag:

```
int which;
which = [myView tag];
```

All Views can respond to a **tag** message. If a subclass doesn't implement its own method, it inherits the version defined in View, which always returns -1.

You'd use the **tag** method when you know the **id** of an object, but not its tag. It's also possible to find a View's **id** from its tag.

```
id theView;
theView = [self findViewWithTag:41];
```

Chapter 7

Program Dynamics

7-5	Event Handling
7-6	Setting Up Event-Handling Objects
7-9	setUp() Example
7-12	Opening Files
7-13	Final Initialization
7-14	Event Masks
7-15	Asking for Particular Events
7-16	Receiving Keyboard Events
7-16	Receiving Timer and Cursor-Update Events
7-17	Selecting an Application, Window, and View
7-18	The First Responder
7-19	Changing the First Responder
7-20	Notification
7-21	Accepting First Responder Status
7-21	The Key Window and Main Window
7-22	Changing the Key Window
7-23	Notification
7-24	The Active Application
7-25	Changing the Active Application
7-26	Notification
7-27	Event Messages
7-29	Keyboard Events
7-29	Left Mouse Events
7-30	Hit Testing
7-30	Trapping the First Event
7-31	Selecting the First Responder
7-32	Right Mouse Events
7-32	Mouse-Exited and Mouse-Entered Events
7-34	Kit-Defined Events
7-34	Window-Moved
7-34	Window-Exposed
7-35	Window-Resized
7-35	Application-Defined Events
7-36	System-Defined Events
7-36	Cursor Coordinates
7-37	Querying the Cursor
7-37	Testing the Cursor's Location
7-39	Event Messages in the Responder Chain
7-40	Action Messages
7-41	The Target
7-42	The Action

7-43	Action Messages in the Responder Chain
7-45	Kit-Defined Action Methods
7-47	The Tag
7-48	The Sender as an Argument
7-49	Keyboard Alternatives
7-49	From Key Down to Key Equivalent
7-50	Command Key-Down Events
7-50	commandKey: Messages
7-51	Initiating performKeyEquivalent: Messages
7-51	Passing performKeyEquivalent: Messages
7-51	Unhandled Messages
7-52	View Methods
7-53	Changing the Cursor
7-53	Defining a Cursor
7-54	Cursor Rectangles
7-55	Registering and Resetting Cursor Rectangles
7-56	Wait Cursors
7-57	Modal Event Loops
7-58	Coordinating Mouse Events
7-60	Getting and Peeking at Events
7-61	Scheduling
7-63	Using Timer Events
7-64	Timer Example
7-65	Avoiding Spin Loops
7-66	Modal Windows
7-66	Breaking the Loop
7-67	Return Codes
7-67	Modal Sessions
7-69	Drawing in the View Hierarchy
7-70	View Coordinate Systems
7-71	Flipping the Coordinate System
7-73	Transitivity
7-73	Unflipping
7-74	Drawing Text
7-74	Notifying Superviews
7-75	Modifying Default Coordinates
7-76	Rotated Bounds
7-78	Transitivity
7-78	Basic and Temporary Coordinate Systems
7-79	Converting Coordinates
7-80	Focusing on a View
7-80	How Focusing Works
7-81	Clipping
7-82	Using the Superview's Coordinates
7-83	Modifying drawInSuperview Coordinates
7-83	Flipping drawInSuperview Coordinates
7-85	Locking and Unlocking the Focus
7-86	Drawing Methods

7-86	Drawing Rectangles
7-88	Graphics State Parameters
7-90	The Display Methods
7-91	Order of Display
7-92	Display Method Arguments
7-93	Displaying Background Views
7-94	Registering as an Opaque View
7-94	Finding the Background View
7-95	Window's Display Methods
7-95	Automatic Display Messages
7-96	Managing a Window's Display
7-96	Updating Windows
7-96	On-Screen Windows
7-97	Updating Menus
7-98	Updating Views
7-99	Suspending Display
7-100	Suspending flushWindow
7-101	Displaying If Needed
7-102	Modifying the Frame Rectangle
7-103	Resizing Subviews

7-106 Printing

7-106	Generating PostScript Code
7-107	Application Kit Printing Architecture
7-109	Pagination
7-109	Image Placement on the Page
7-110	Display PostScript Contexts
7-110	Panels

Chapter 7

Program Dynamics

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/ReleaseNotes/AppKit.rtf](#)
[/NextLibrary/Documentation/NextDev/ReleaseNotes/AllocInitAndNew.rtf](#)

The previous chapter, “Program Structure,” described how a program based on the Application Kit is constructed from a small number of interconnected objects. This chapter shows how applications use that structure to carry out essential interactive activities:

- Responding to events
- Drawing on the screen (and on the printed page)
- Sending and receiving remote messages

Because events motivate almost all program activities, the chapter begins with a discussion of event handling in the Application Kit.

Event Handling

NeXT applications are driven by the user’s actions on the keyboard and mouse—that is, by events. The application receives an event from the Window Server, responds to it, then looks for the next event. If there is no next event, the application waits until the user does something and an event is received.

The Application object, NXApp, initiates this event loop—the application’s *main event loop*—when it receives a **run** message:

```
[NXApp run];
```

On each cycle of the loop, NXApp gets an event, analyzes it, and sends an appropriate message to initiate the application’s response. It passes keyboard events to the key window and mouse events to the Window associated with the event in the event record. The Window, in turn, dispatches the event to one of its Views. Most of your program’s activity will be in reaction to the messages NXApp and the application’s Windows send out after receiving an event.

- Some of their *event messages* will reach objects that can respond directly.

- Some event messages will reach Control objects—Buttons, Sliders, Scrollers, TextFields, and the like—and will be translated into more specific *action messages* for other objects.

NXApp continues to get events out of the event queue and dispatch them until the event loop is broken. Typically, it's broken only when the application terminates. If the response to an event includes a **terminate:** message,

```
[NXApp terminate:self];
```

NXApp closes all the application's windows, frees its objects, and exits the program.

Note: The **terminate:** method takes an argument only so that it can respond to an action message, usually an action message coming from the Quit command. It doesn't actually look at its argument, so it doesn't matter what value is passed. See “Action Messages,” later in this chapter, for more on the structure of these messages.

Sometimes an application's response to an event is to set up a *modal event loop* that will get all subsequent events for a short period of time. For example, the response to a mouse-down event may be to set up a modal loop that will collect events until the user releases the mouse button and a mouse-up event is received. Modal loops are set up within the main event loop in response to an event or action message.

Most of the unique behavior of your application will be encoded in class definitions for objects that can respond to event and action messages.

This section of the chapter looks at event handling in a typical application, beginning where the application itself begins, with the code that sets up its event-handling objects.

Setting Up Event-Handling Objects

An Objective-C program begins just as a C program does, by calling its **main()** function. In a program based on the Application Kit, **main()** is usually very short. Its job is to set up the Application object and other core objects your program needs at startup, then to turn over control of the program to them. It can be as short as just three or four lines of code:

```
main()
{
    [Application new];
    setUp();
    [NXApp run];
    [NXApp free];
}
```

In this version of **main()**, the **new** method creates an Application object, NXApp, which receives a **run** message to begin getting events from the Window Server. When the **run** method quits, **free** cleans up after the application.

Most of the application's time is spent in the **run** method, getting and responding to events. But before a **run** message can be sent, the application must prepare itself for the events it's about to receive. It must:

- Create the Windows, Views, and other objects it needs to handle events at startup. It's possible to create new objects at any time while the program is running, but an initial set of core objects must be in place before the first event is processed.
- Initialize the objects so they're connected into a program framework.
- Present the application to the user by placing the initial display on-screen and designating a Window to serve as the initial key window. If the application allows users to edit text or graphics within the window, it should also designate a View to show the initial selection, and then set the selection.

In the version of **main()** shown above, all this code has been segregated into the **setUp()** function. For some applications, it might be appropriate to define a subclass of the Application class and include setup code in a redefined version of the **new** method:

```
@implementation MyApplication : Application
{
}
+ new

{
    self = [super new];
    /* setup code goes here */
}
```

The **main()** function could then consist of just a single line of nested messages:

```
main()
{
    [[[MyApplication new] run] free];
}
```

When you design your application using Interface Builder, you can do most of the work of setting up your application by selecting objects from palettes and editing them on-screen. Interface Builder lets you graphically lay out the Windows, Views, and other interface objects your application needs, initialize them, display them, and archive them in a file for later use. The file can then be inserted into the **__NIB** segment of the application executable:

```
cc ... -segcreate __NIB myProject.nib myProject.nib
```

Here the **-segcreate** linker option copies the **myProject.nib** file into a section of the **__NIB** segment and assigns the section the same name as the file.

A message to NXApp to open this section takes the place of the **setUp()** function:

```
main()
{
    [Application new];
    [NXApp loadNibSection:@"myProject.nib" owner:NXApp];
    [NXApp run];
    [NXApp free];
}
```

In the example above, the objects that were archived in **myProject.nib** are loaded into memory and connected to the object that “owns” them, NXApp.

Archived objects can have owners other than NXApp:

```
main()
{
    id theHub;

    [Application new];
    theHub = [MyCoordinator new];
    [NXApp setDelegate:theHub];
    [NXApp loadNibSection:@"newProject.nib" owner:theHub];
    [NXApp run];
    [NXApp free];
}
```

For this example, the programmer defined a class, **MyCoordinator**, to contain the basic algorithms of the application—its inner workings as opposed to its interface. After creating an Application object, **main()** creates an instance of the **MyCoordinator** class, makes it the delegate of the Application object, and connects it with the user-interface objects archived in **newProject.nib**. Whenever a Window object is loaded from the archive, it’s automatically added to NXApp’s list of windows; NXApp doesn’t have to be named as the Window’s owner.

With Interface Builder, you can create any number of archive files (or **__NIB** sections), each with a different set of objects and, if desired, a different owner. For example, every panel the application uses could be archived separately. The archive file (or section) would contain the Panel object and all the Control objects it displays; its owner would be an object you’d design to receive action messages from the panel and coordinate its activities. The owner could be made the Panel’s delegate; if needed, NXApp could be provided with an instance variable to keep track of the Panel’s owner.

In this way, your application can build its own network of objects, all relying on the basic network of core objects described under “Program Framework” in the previous chapter.

Interface Builder is described in the next chapter. This chapter concentrates on the program structure defined in the Application Kit, and so returns to a simple version of a **setUp()** function that can be bracketed (as can **loadNibSection:owner:**) by **new** and **run** messages.

setUp() Example

Interface Builder is the preferred way to program an application. But to show the steps required by the Application Kit for setting up an application, the code for a simple program is listed below.

There are three windows in this example program—a main menu, an information panel, and a small window where the user can enter and edit text. All three are illustrated in Figure 7-1.

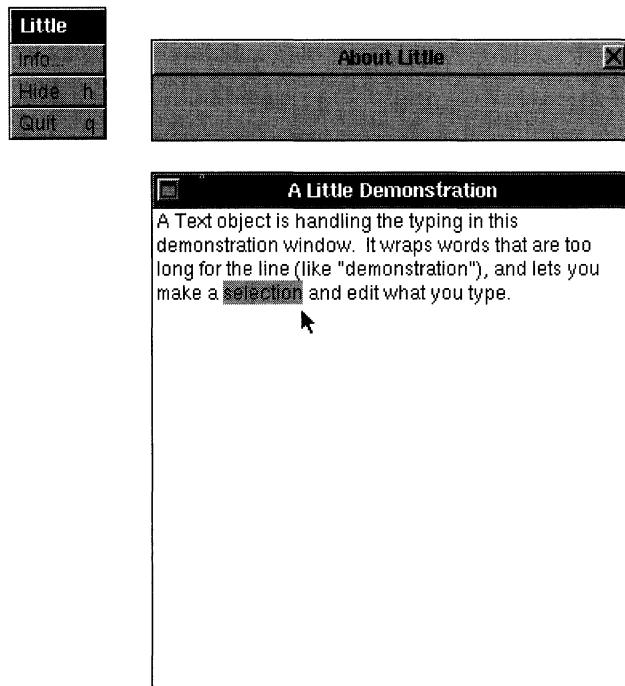


Figure 7-1. Little

This program has the basic elements of a real application, but is too simplified to be very useful, hence its name, “Little.” But despite its simplicity, it behaves like a full-fledged application. It can be hidden, its principal window can be miniaturized, the panel and menu disappear when it’s deactivated and reappear again when it’s activated, the keyboard alternatives work, the text the user types can be selected and edited, and so on.

So that you can compile this little program and try it out for yourself, the source code and a makefile for it are on-line in [/NextLibrary/Documentation/NextDev/Examples/Little](#).

The entire application is written in two functions, **main()** and **setUp()**; it includes no class definitions of its own. The **main()** function has just four lines of code, as illustrated above and repeated below; the **setUp()** function is just 58 lines, so the whole program can be printed in a little over a page.

```

#import <appkit/appkit.h>

void setUp(void)
{
    id      myWindow, myPanel, myMenu, windowText;
    NXRect  aRect;

    /*** Step 1: Set up a Window ***/
    NXSetRect(&aRect, 100.0, 350.0, 300.0, 300.0);
    myWindow = [Window newContent:&aRect
                           style:NX_TITLEDSTYLE
                           backing:NX_BUFFERED
                           buttonMask:NX_MINIATURIZEBUTTONMASK
                           defer:NO];
    [myWindow setTitle:@"A Little Demonstration"];

    NXSetRect(&aRect, 0.0, 0.0, 300.0, 300.0);
    windowText = [Text newFrame:&aRect
                           text:@""
                           alignment:NX_LEFTALIGNED];
    [windowText setOpaque:YES];
    [[myWindow contentView] addSubview:windowText];

    /*** Step 2: Set up a Panel ***/
    NXSetRect(&aRect, 100.0, 700.0, 300.0, 40.0);
    myPanel = [Panel newContent:&aRect
                           style:NX_TITLEDSTYLE
                           backing:NX_BUFFERED
                           buttonMask:NX_CLOSEBUTTONMASK
                           defer:YES];
    [myPanel setTitle:@"About Little"];
    [myPanel removeFromEventMask:(NX_KEYDOWNMASK | NX_KEYUPMASK) ];

    /*** Step 3: Set up a Menu ***/
    myMenu = [Menu newTitle:@"Little"];
    [[myMenu addItem:@"Info..."
                  action:@selector(orderFront:)
                  keyEquivalent:'\0'
                  setTarget:myPanel];
    [myMenu addItem:@"Hide"
                  action:@selector(hide:)
                  keyEquivalent:'h'];
    [myMenu addItem:@"Quit"
                  action:@selector(terminate:)
                  keyEquivalent:'\0'];
    [myMenu sizeToFit];
    [NXApp setMainMenu:myMenu];

    /*** Step 4: Display all windows that aren't deferred ***/
    [myWindow display];

    /*** Step 5: Move myWindow on-screen ***/
    [myWindow orderFront:nil];
}

```

```

    /*** Step 6: Make it the key window ***
    [myWindow makeKeyWindow];

    /*** Step 7: Show a selection in the key window ***
    [windowText selectAll:nil];
}

main()
{
    [Application new];
    setUp();
    [NXApp run];
    [NXApp free];
}

```

All the methods used in this example are defined in the Application Kit. Some were discussed in the previous chapter under “Managing Windows”; others are described in Chapter 9, “User-Interface Objects.” The **NXSetRect()** function, which assigns values to an NXRect structure, is discussed in *NeXTstep Reference, Volume 2*.

As its first act (step 1), the **setUp()** function creates a new instance of the Window class and titles it. It then creates a Text object the same size as the Window’s content area and makes it a subview of the Window’s content view. (It could equally as well have made the Text object the content view and freed the default content view provided by the Window object.)

Next (step 2), **setUp()** creates a Panel to serve as Little’s information panel. The Panel is assigned a title, and keyboard events are removed from its event mask so that it can’t become the key window. This panel will behave just like any other information panel, except for the glaring fact that it doesn’t contain any information. Additional code would be required to draw the application’s icon in the panel and provide text giving version and copyright information. (Little is too little to have any such information to impart.) Usually, information panels are designed in Interface Builder.

The **setUp()** function then (step 3) creates a menu with the three minimal commands every application should have:

- The Info command sends an **orderFront:** message to its target, the information panel, placing the panel on-screen at the front of its tier.
- The Hide command sends a **hide:** message that NXApp will respond to by hiding all the application’s windows. The command is assigned Command-h as its keyboard alternative.
- The Quit command sends a **terminate:** message that NXApp will respond to by shutting down the application. Although it’s not required by the user-interface guidelines, the command is assigned the common keyboard alternative Command-q.

The size of the Menu is altered so that it exactly fits the three commands, and it’s made the main menu of the application.

At this point, all the objects in the application have been created and initialized. It remains only to present the application to the user by displaying the contents of the one window that hasn't been deferred (step 4), moving the principal Window of the application on-screen (step 5), making it the key window on launch (step 6), and setting the Text object to display an initial selection (step 7). Steps 5 and 6 could have been combined by using a single method, **makeKeyAndOrderFront:**.

The **display** message that **myWindow** receives in step 4 calls upon every View within its view hierarchy to draw itself. The images are displayed into the window's backup buffer, but nothing appears on-screen. A window must be reordered into the screen list for its display to be visible. That's the function of step 5.

It's unnecessary to supply code in step 4 to display **myMenu** and **myPanel** along with **myWindow**. Menus are created as deferred windows and so are displayed automatically just before they're placed on-screen. The Panel was also created as a deferred window in step 2, and will be displayed just before the user brings it to the screen with the Info command.

Similarly, no code is needed to put the Menu and Panel on-screen. When the application is activated, the Application Kit places the main menu at the location specified by the NXMenuX and NXMenuY parameters. By default, it's in the upper left corner of the screen. The Info command will put the information panel on-screen and the panel's close button will remove it again.

Opening Files

If the Little program had the ability to open and display files, it would be organized a bit differently. Instead of creating a main window (**myWindow**) at the outset, it would have waited for an instruction indicating which file to open. Only then would it create a Window for the file and, in actions paralleling steps 4, 5, 6, and 7 of the example program, display the file in the Window, move the Window on-screen, make it the key window, and designate an initial selection.

The instruction to open a file can come from three different sources:

- If the user launches an application by double-clicking an icon for one of its files, the Workspace Manager passes the name of the file to the application. The Application object receives an **openFile:ok:** message just before getting its first event.
- If the user double-clicks a file in a directory window after the application is launched, the Workspace Manager sends the application a message with the name of the file. Application's **openFile:ok:** method is again entrusted with the message.
- If the user selects a file from the application's Open panel, the application must get the name (and directory) of the file from the OpenPanel object and open the file. The Open panel is brought to the screen by the Open menu command, in much the same way that an information panel is.

In all three cases the application must supply the code that creates the Window, displays the file, places the Window on-screen, makes it the key window, and sets the initial selection within the window. But instead of doing this as part of a `setUp()` function, it's done in response to the user's selection of a file.

If the selection of a file generates an `openFile:ok:` message to the Application object, you should put this code in an `appOpenFile:type:` method defined in either an Application subclass or as part of the Application object's delegate. `openFile:ok:` does some preprocessing of the message, and sends an `appOpenFile:type:` message for the delegate or NXApp to actually open the file.

Final Initialization

Just before the `run` method gets its first event, it does some final initialization to make sure that the application is ready:

- It activates the application, which will cause it to receive an application-activated subevent (of the kit-defined event).
- It provides the application with default Listener and Speaker objects, if a Listener and Speaker weren't created in the application's setup code. These objects give the application the ability to communicate with the Workspace Manager and other applications.
- It checks in the application with the network name server so that it will have a public port where it can receive messages from the Workspace Manager and other applications. The application is checked in under the name returned by the Application object's `appListenerPortName` method, which is usually the name assigned in the header to the `_ICON` segment of the application executable, or possibly the name passed in the first string of the `NXArgv` array. You can override `appListenerPortName` so that it returns a different name. If it returns `NULL`, the application is assigned a private port so that it can communicate with the Workspace Manager, but it won't have a public port.
- It sends an `openFile:ok:` message to NXApp, if the user launched the application by double-clicking a file icon.

The `run` method also gives the application one last chance to do any final initialization of its own. It sends the Application object's delegate an `appDidInit:` message, if the delegate has a method that can respond.

The delegate's `appDidInit:` method can make any adjustments necessary for the events that are about to arrive. For example, if an application normally opens and displays a file, but the user launched it by double-clicking an application icon rather than a file icon, `appDidInit:` could open an empty window for the user to work in.

The argument passed in an `appDidInit:` message is the Application object's `id`.

Event Masks

Each window's event mask, maintained by the Window Server, determines which events the Server can send to the application process for that window. See Chapter 5, "Events," for information on the structure of event masks and the methods used to associate events with particular windows.

Each Window object keeps track of its own event mask through its **winEventMask** instance variable. The **eventMask** method returns the current mask:

```
int myMask;  
myMask = [myWindow eventMask];
```

A new menu or list has these event types in its default event mask:

mouse-down	(both left and right)
mouse-up	(both left and right)
mouse-dragged	(both left and right)
kit-defined	

The default mask for panels and standard windows includes this full range of events:

key-down	
key-up	
mouse-down	(both left and right)
mouse-up	(both left and right)
mouse-entered	
mouse-exited	
kit-defined	
system-defined	
application-defined	

Minindows and icons, whether in or out of the dock, have a similar event mask, but exclude key-down and key-up events.

To ensure that they work properly, you should refrain from changing the event masks of menus, lists, icons, and minindows. However, the event masks of standard windows and panels can be altered to suit the needs of your application.

If you want a window to receive an event type that's not included in the default mask, or to avoid receiving an event type that is included, you must change the Window object's event mask:

```
int oldMask;  
oldMask = [myWindow setEventMask:myNewMask];
```

If **myNewMask** is different from the mask maintained by the Window Server for **myWindow**, **setEventMask:** changes both the Window Server's mask and **myWindow**'s

winEventMask instance variable. The following code resets a default mask to include flags-changed events:

```
oldMask = [myWindow setEventMask:([myWindow eventMask] |  
                                NX_FLAGSCHANGED)];
```

This operation could be accomplished more simply with the **addToEventMask:** method:

```
oldMask = [myWindow addToEventMask:NX_FLAGSCHANGED];
```

There's also a method for removing event types from the current mask:

```
oldMask = [myWindow removeFromEventMask:NX_APPDEFINEMASK];
```

Each of these methods returns the former event mask so that you can cache it and restore it later if needed.

Asking for Particular Events

An application should limit the events it receives from the Window Server to just those it's interested in. This saves processing time, reduces the amount of code you must write, and limits potential errors from handling unwanted events.

But, for most types of events, it's best to set the window's event mask at the outset so that it will receive all the events you ever want for it. It's generally not a good idea to change the mask in response to events, since the user might act between the time you send the message and the time the Window Server gets around to resetting the mask. The delay could cause your application to miss events it expected to receive.

The principal exception to this rule arises when you want to receive mouse-dragged or mouse-moved events:

- Since these events are sent continuously, as long as the mouse is in motion, it doesn't matter that the Window Server won't begin sending them until it resets the event mask. Missing the first in a series of mouse-dragged or mouse-moved events is usually of little consequence.
- Since dispatching a continuous stream of events demands a lot of processing time, you should keep the mask for these two events set only for a limited period. Have your application set the Window's event mask to include mouse-dragged or mouse-moved events just before it's ready to respond to them and have it reset the mask when it's finished. Your application shouldn't ask for these events and then attend to other things.

A menu includes mouse-dragged events in its event mask because it must always be ready to respond to them. When using a menu, users typically drag through the list of menu commands.

Receiving Keyboard Events

The Application Kit guarantees that keyboard events are sent to the active application, regardless of whether its on-screen windows have event masks that accept keyboard events. This enables the application to respond to keyboard alternatives when it's active, even if it has no windows for the user to type in.

Although the event mask doesn't determine which application gets keyboard events, it does help determine which Window within the application will receive them:

- The Application Kit associates keyboard events with the current key window.
- For a Window to be the key window, it must have an event mask that accepts key-down events. The initial event mask for all Windows (except menus, miniwindows, and icons) includes key-down events, so, by default, they all are potential key windows.

Since the NeXT user interface requires every standard window to be the key window whenever it's the main window the user is working in, standard windows should keep key-down events in their event masks. If the window doesn't display typing, it will beep whenever it receives a key-down event.

However, if you have a panel that won't respond to keyboard events and shouldn't be marked as the key window, you must reset its mask to exclude key-down events. Any Window that excludes key-down events should also exclude key-up events, so this example removes both event types from the event mask:

```
[myPanel removeFromEventMask: (NX_KEYDOWNMASK | NX_KEYUPMASK)];
```

A Panel should remove keyboard events from its event mask if it meets all three of these tests:

- It doesn't display typing.
- It's not an attention panel.
- It doesn't have a button that the user can operate from the keyboard by pressing Return. Normally, such a button is only permitted in an attention panel or in a panel that displays a text field.

Receiving Timer and Cursor-Update Events

A Window's event mask never needs to include either timer events or cursor-update events.

Timer events are synthetic events, generated by the application when it needs them. Because they aren't sent across the connection from the Window Server, the event mask doesn't determine whether they can be received. See "Using Timer Events" under "Modal Event Loops" later in this chapter for information on how to generate these events.

Cursor-update events signal that it's time to change the cursor image. Applications don't respond to these events directly; the change is made by the Application Kit.

A Window receives cursor-update events if a cursor has been associated with a particular area (a cursor rectangle) located within the window and registered with the Window object. The Kit makes sure the application gets these events when they're needed; the event doesn't have to be included in the Window's event mask. However, because cursor-update events are based on mouse-entered and mouse-exited events, it's best to keep those two events in the event mask if you want to receive cursor-update events.

See "Changing the Cursor" later in this chapter for more information on setting cursor rectangles.

Selecting an Application, Window, and View

After the application's core objects have been set up, the event masks of its windows have been adjusted, and a **run** message sent, the Application object begins getting events from the Window Server. Most events are dispatched in messages to other objects, as described in the next section, "Event Messages."

Left mouse-down events also serve to select the application, the window, and even the View that will be the focus of future events. The selected object's event-handling status is designated by terms that are mostly familiar from the user interface:

<i>active application</i>	The application that's been selected to receive keyboard events, and to have visible menus and panels.
<i>key window</i>	The Window that's been selected to handle keyboard events for the application, and to be the primary recipient of action messages from menus and panels.
<i>main window</i>	The Window that's the principal focus of user actions. It's usually identical to the key window.
<i>first responder</i>	The View that's been selected to have the first chance at responding to keyboard events and action messages sent to a Window.

The active application, key window, and main window are concepts important to the user interface and are defined in Chapter 2, "The NeXT User Interface." The first responder is no less important, but, like all Views, is a part of the implementation of the user interface, rather than part of its definition.

The First Responder

When the user clicks in a Text object, such as the one in the Little example program listed earlier under “Setting Up Event-Handling Objects,” it’s selected to receive subsequent events, especially keyboard events. The click also selects the insertion point where future typing will appear.

That simple demonstration program had just one View—a Text object—displayed within its Window. But imagine a Window, such as the one illustrated in Figure 7-2, with four Text objects sharing its content area.

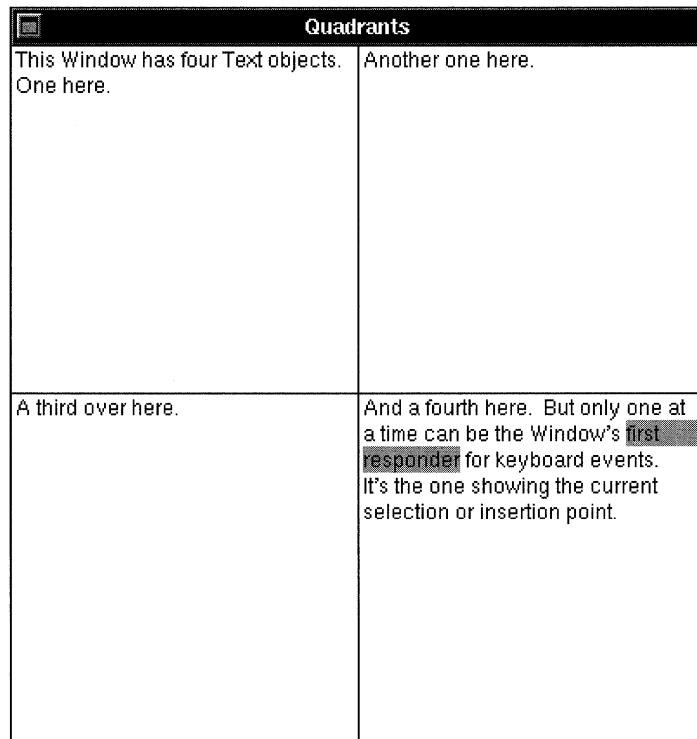


Figure 7-2. Four Text Objects in a Window

By clicking in one quadrant or another, the user determines where typing will appear (which Text object will receive keyboard events). Only one of the four objects will show a selection or insertion point at a time.

The object that’s selected to be the focus of future events for a Window is the *first responder*. Each Window has its own first responder, which it returns when asked:

```
id handler;
handler = [myWindow firstResponder];
```

The first responder is typically a View object in the Window’s view hierarchy, but it can be any Responder. At the outset, each Window is its own first responder. Because Windows generally can’t respond to keyboard and mouse events, this is usually the same as having a **nil** first responder.

The first responder is a central actor in the handling of event and action messages. It receives:

- Keyboard event messages (**keyDown:**, **keyUp:**, and **flagsChanged:**).
- Action messages from Controls that don’t have explicit targets of their own. This includes messages from menu commands that affect the current selection, such as the Cut, Copy, Paste, Bold, and Italic commands. See “Action Messages,” later in this chapter.
- Messages that notify the first responder when the Window becomes the key window and when it stops being the key window (**becomeKeyWindow** and **resignKeyWindow**). See “The Key Window and Main Window” below.
- Mouse-moved event messages (**mouseMoved:**).

If the first responder can’t respond to any of these messages, its next responder is given a chance to respond. See “Event Messages in the Responder Chain” and “Action Messages” later in this chapter for details.

Changing the First Responder

As shown by the example illustrated in Figure 7-2 above, the Application Kit lets the user pick the first responder; the Window object alters its **firstResponder** instance variable on the basis of the left mouse-down events it receives (see “Left Mouse Events,” under “Event Messages” below).

Before making the View selected by a mouse-down event the first responder, the Window sends it an **acceptsFirstResponder** message to ask whether it accepts this role. By default, all Views—in fact, all Responders—answer NO, leaving the current first responder in place. An object can agree to be made the first responder simply by implementing an **acceptsFirstResponder** method that answers YES:

```
- (BOOL)acceptsFirstResponder
{
    return YES;
}
```

Some objects may return YES under certain circumstances and NO under others. If a Text object displays editable or selectable text, it answers YES. If the text is neither editable nor selectable, it answers NO.

If the selected View returns YES, the Window attempts to make it the first responder through its **makeFirstResponder:** method:

```
[self makeFirstResponder:selectedView];
```

As used by the Application Kit, the **acceptsFirstResponder** and **makeFirstResponder:** methods permit users to alter the first responder in a carefully regulated manner. You can also set the first responder from within your application using the same methods. This is most appropriate when registering an object as the Window's initial first responder on launch.

To function as the initial first responder of a Window, a Text object needs not only to be made the first responder, it must also be assigned a selection. The Text class defines three methods that do both; they register a new selection and send the Window a **makeFirstResponder:** message. The **selectAll:** method used in the Little example earlier in this chapter selects all the object's text; the **selectText:** method does the same. The **setSel::** method defines a range of text to select. In the example below, it selects the characters at positions 52 through 190 (up to position 191):

```
[windowText setSel:52 :191];
```

If the range of text selected is 0—if both arguments to **setSel::** are the same, or there's no text for **selectAll:** and **selectText:** to select—the selection is an insertion point. The **selectAll:** message in the Little example will make the receiving Text object show a blinking caret for the insertion point when the program is launched and the Window becomes the key window; it won't take an initial click to select the Text object as the first responder.

Notification

The **makeFirstResponder:** method first sends the current first responder a **resignFirstResponder** message to notify it that a change is about to be made; the new first responder is then notified with a **becomeFirstResponder** message. The default implementation for both these methods is simply to return **self**. An object can override the default to keep track of whether it's the current first responder, or to prevent the change from being made:

- If an object returns **nil** to a **resignFirstResponder** message, it refuses to be deactivated and remains the first responder. No **becomeFirstResponder** message is sent and **makeFirstResponder:** returns **NULL**.
- If an object returns **nil** to a **becomeFirstResponder** message, it refuses to be the new first responder. Since the current first responder has already resigned, the Window is made the first responder instead. This returns the Window to its state before any events or **makeFirstResponder:** messages were received.

An object could refuse to become the first responder if its internal state temporarily prevents it from responding to events and action messages. An object might refuse to give up being the first responder if it needs to receive additional events. For example, an object that asks the user to type a directory name might remain the first responder until it receives the name of a valid directory.

Accepting First Responder Status

An object should agree to be the first responder (by returning YES to an **acceptsFirstResponder** message) only if it needs to receive some of the event and action messages that are directed to a first responder. These messages were listed above and include, most prominently:

- Keyboard event messages
- Action messages that aren't hard-wired to a specific target

Action messages from the Controls within the Font panel and from menu commands such as Cut, Copy, and Paste fit this description. They affect the current selection, which can change from Window to Window and from View to View. More precisely, they affect the current selection of the first responder in the key window or main window.

Any View that displays material that the user can select or edit must be able to respond to untargeted action messages like these, and therefore must accept first responder status. In the Application Kit, for example, Text objects, TextFields, and Forms agree to be the first responder when they display editable text, whereas Buttons, Sliders, and Scrollers always refuse.

Most objects that refuse to be the first responder fit into one of two categories:

- If an object responds only to mouse events, it can set up its own modal loop to get all the events it needs, except the mouse-down events that initiate the loop. It doesn't have to be the first responder to receive mouse-down event messages. See "Modal Event Loops," later in this chapter, for a description of this type of object.
- If an object inherits from View simply so that it can draw on the screen, not so that it can respond to events, it won't want any events at all. The event messages it receives will be passed on to its next responder.

The Key Window and Main Window

The Application Kit changes the key window (and main window) in response to left mouse-down events received from the Window Server. If the window associated with the event isn't already the key window, it's made the new key window provided that it has an event mask that accepts key-down events. Unless the window is a panel, it's also made the main window. The main window changes only when the key window does.

Each Window object keeps track of whether it's the key window or main window:

```
BOOL keyStatus, mainStatus;  
  
keyStatus = [myWindow isKeyWindow];  
mainStatus = [myWindow isMainWindow];
```

The Application object can identify which of its Windows is the key window and main window:

```
id key, main;  
  
key = [NXApp keyWindow];  
main = [NXApp mainWindow];
```

These two methods return **nil** if the application isn't active or if there is no key window or main window.

Changing the Key Window

You can alter the key window (and main window) programmatically with the **makeKeyWindow** method:

```
[myWindow makeKeyWindow];
```

This method can also be performed indirectly, through the **makeKeyAndOrderFront:** method. **makeKeyAndOrderFront:** combines a **makeKeyWindow** message with an **orderFront:** message.

```
[myWindow makeKeyAndOrderFront:self];
```

For the Window that receives a (direct or indirect) **makeKeyWindow** message to actually be made the key window, it must be on-screen and must accept key-down events. For it also to be made the main window, it must be on-screen, accept key-down events, and not be a Panel. The **canBecomeMainWindow** method returns whether these three conditions are true:

```
BOOL potentialMain;  
potentialMain = [myWindow canBecomeMainWindow];
```

Since the key window and main window belong only to the active application, the receiving Window's application must also be the active application when a **makeKeyWindow** message is sent. If it's not, the message serves only to register the intended key window (and main window) for the next time the application is activated. The user can override this intention simply by clicking another window.

Programmatically making a Window the key window is appropriate only in a limited number of situations:

- After the user acts in a panel that's the key window, the panel should return key-window status to the main window. For example, when the Font panel sets the font of the current selection, it makes the main window (where the selection is located) the key window.
- To respond to a remote message, the application might need to designate an appropriate key window. After receiving a message to open a file, for example, the application should make the window that displays the file the key window.
- An initial key window is designated in the application's setup code. For example, the Little demonstration program listed under "Setting Up Event-Handling Objects" above sent a **makeKeyWindow** message to its principal Window as part of its **setUp()** function. Since the application wasn't active at the time the message was sent, it served only to register the Window as the desired key window (and main window) once the application was activated.

Notification

When the key window changes, the new key window is notified with a **becomeKeyWindow** message. If the Window also becomes the main window, it's notified with a **becomeMainWindow** message. The former key window and main window are notified of their lost status with **resignKeyWindow** and **resignMainWindow** messages:

```
[thatWindow resignKeyWindow];
[thisWindow becomeKeyWindow];

[thatWindow resignMainWindow];
[thisWindow becomeMainWindow];
```

These messages are sent whenever the key window or main window changes, no matter what the reason. The change could be caused by:

- A left mouse-down event deactivating one application and activating another
- A left mouse-down event within another window of the same application
- The user hiding an application, and thus deactivating it
- The user unhiding an application, and thus activating it
- The user closing the key window or main window
- The user miniaturizing the key window or main window
- A **makeKeyWindow** message in the active application

In other words, the messages exactly parallel the changes in window status that the user sees. See Chapter 2 for more on how user actions affect a window's status.

When any of the four messages are generated by a left mouse-down event, they're sent before any **becomeFirstResponder** and **resignFirstResponder** messages that might be generated by the same event.

The Window class defines methods that can respond to all four messages shown above. Each method records the Window’s change in status and notifies the Window’s delegate of the change, provided the delegate has a method that can respond to the message:

```
[[self delegate] windowDidBecomeKey:self];
[[self delegate] windowDidResignKey:self];

[[self delegate] windowDidBecomeMain:self];
[[self delegate] windowDidResignMain:self];
```

The Window also passes **becomeKeyWindow** and **resignKeyWindow** messages on to its first responder, if the first responder can respond. In the Application Kit, the Text object uses these messages to learn when to begin blinking the caret marking the insertion point and when to stop. The caret should blink only in the Text object that will display the user’s typing—that is, only in the first responder of the key window.

Your application can use these notification messages to keep itself current with the user’s actions. For example, you may want to make sure that the PrintInfo object cached by the Application object reflects the document in the key window. You might also want to update panels (such as an “inspector” panel) so that they display information appropriate for the main window.

To take advantage of the notification, you must implement methods that can respond either to the messages sent to the Window or to the ones sent to its delegate. The methods you define in a Window subclass should perform the default versions defined in the Window class:

```
- becomeKeyWindow
{
    [super becomeKeyWindow];
    .
}
```

The Active Application

When a left mouse-down event selects a new active application, it generates an application-activated subevent (of the kit-defined event). The subevent is sent to the application even before the mouse-down event.

When one application is activated, another might be deactivated, so a mouse-down event may also cause an application-deactivated subevent (also of the kit-defined event) to be sent to the current active application. The active application is always deactivated before another application is activated, so the application-deactivated subevent is sent before the application-activated subevent. But since they’re sent to different processes, it’s not determined which one will be acted on first.

When the user hides an application, an application-deactivated subevent is generated, but no application-activated subevent.

The **isActive** method returns the application's current status:

```
BOOL activeStatus;
activeStatus = [NXApp isActive];
```

The **activeApp** method returns a user object (an integer) identifying the PostScript execution context of the active application:

```
int activeOne;
activeOne = [NXApp activeApp];
```

Changing the Active Application

Usually, applications are activated by left mouse-down events that spawn application-activated and application-deactivated subevents. But the Application Kit is sometimes called upon to activate an application in the absence of an event. An application is activated when:

- It's launched.
- It receives a message to open a file.
- It's returned to the screen after being hidden.

In the first two cases, the activation is conditional: The application will become active only if there's no current active application. This condition is usually met because the Workspace Manager deactivates the current active application both when it launches a new application and when it sends an inactive application an **openFile:ok:** message. The condition won't be met only if the user chooses another application to work in before the target application has a chance to activate.

If your application accepts messages (other than **openFile:ok:** messages) from other applications, it may need to activate itself in response:

```
[NXApp activateSelf:NO];
```

The NO flag indicates that the activation is conditional. This method may generate application-activated subevents. With YES as an argument, the activation is unconditional; it will force the current active application to be deactivated and so may also generate application-deactivated subevents for another application.

In general, protocols between cooperating applications require the application receiving a message to activate itself conditionally, if it needs user interaction to do its work. The sending application should deactivate itself so the receiving application can become active:

```
[NXApp deactivateSelf];
```

If, in response to a message, an application activates itself unconditionally, it should restore the previous active application (the one sending the message) when it's finished. The **activateSelf:** method returns the PostScript execution context of the previous active application:

```
int lastActive  
lastActive = [NXApp activateSelf:YES];
```

The number can be used to reactivate the application later:

```
[NXApp activate:lastActive];
```

Instead of using **activateSelf:** for unconditional activation, applications can also use the **unhide:** method:

```
[NXApp unhide:anId];
```

unhide: includes an **activateSelf:** message (with YES as its argument) and also ensures that the windows of the newly activated application aren't hidden.

Notification

When the Application object receives an application-activated subevent from the Window Server, it first does what's necessary to activate the application. It then sends itself a **becomeActiveApp** message. An application-deactivated subevent generates a **resignActiveApp** message.

The methods that respond to these two messages simply notify the Application object's delegate of the change in status, if the delegate has a method that can handle the message:

```
[[self delegate] appDidBecomeActive:self];  
[[self delegate] appDidResignActive:self];
```

The sole purpose of **becomeActiveApp** and **resignActiveApp** messages is to give your application a way of coordinating its activities with changes in its status. You can take advantage of this opportunity either by implementing **appDidBecomeActive:** and **appDidResignActive:** methods for the delegate or by overriding Application's **becomeActiveApp** and **resignActiveApp** methods in a subclass definition.

Event Messages

As the Application object gets events from the Window Server, it dispatches them as Objective-C messages to other objects. With few exceptions, NXApp sends every event to a Window object:

- Keyboard events are sent to the key window. (However, Command key-down events, potential keyboard alternatives, can be sent to any Window.)
- Mouse events are sent to the Window associated with the event—that is, to the Window whose window number is recorded in the **window** component of the event record.
- The window-moved, window-resized, and window-exposed subevents of the kit-defined event are also sent to the Window associated with the event. (The window-resized subevent isn't currently used by the Application Kit. However, should window-resized subevents be posted or placed in the event queue, the Kit would dispatch them to the window in the event record.)

If the event is a keyboard or mouse event, the Window usually sends it on to one of the objects in its view hierarchy. It handles kit-defined subevents itself.

The events that NXApp doesn't dispatch to a Window are:

- Application-defined events, which NXApp handles itself with the aid of its delegate.
- The application-activate and application-deactivate subevents of the kit-defined event. These two subevents are handled internally by the Application Kit; your objects never need to respond to them directly, though they can be notified when the application is activated or deactivated. See “Selecting an Application, Window, and View” above.
- Subevents of the system-defined event. All but the power-off subevent are used internally by objects defined in the Kit; your application never has to deal with them. The power-off subevent is handled by the Workspace Manager, which notifies all the applications it launched that the power is about to go off. Applications generally respond to the Workspace Manager's message rather than to the event itself.
- Timer events. The Application object doesn't dispatch timer events; you should ask for them only when you're prepared to get them out of the event queue yourself. See “Using Timer Events” under “Modal Event Loops” later in this chapter.
- Cursor-update events, which are used internally by the Application Kit. See “Changing the Cursor” later in this chapter.

The object that receives an event from the Application object, or from a Window, gets it in the form of an *event message*—a message to apply a method named after the event type or subtype it reports.

Event Category	Method
Mouse events	mouseDown: (for the left mouse button) mouseUp: (for the left mouse button) mouseDragged: (for the left mouse button) rightMouseDown: (for the right mouse button) rightMouseUp: (for the right mouse button) rightMouseDragged: (for the right mouse button) mouseMoved: mouseEntered: mouseExited:
Keyboard events	keyDown: keyUp: flagsChanged:
Kit-defined subevents	windowExposed: windowMoved: windowResized:
System-defined subevents	powerOff:
Application-defined events	applicationDefined:

In each case, the method takes a single argument, a pointer to the event record. By its very name the event message identifies the event type or subtype; its argument passes along all the other available information about the event.

Note: The Application Kit and this documentation take the point of view of a right-handed user. The primary mouse button, whether left or right, generates “left mouse events” and the other mouse button, if it functions differently than the primary button, generates “right mouse events.” If a user enables the left mouse button to bring the main menu to the cursor, it will generate right mouse events and the right mouse button will generate left mouse events. If neither button is enabled to bring the main menu to the cursor, both buttons generate left mouse events.

The Application Kit implements default versions of methods that respond to event messages. Some have default behavior that your application can inherit and rely on. Others—especially those for keyboard and mouse events—do little or nothing. You must either implement your own methods so that your application can respond to the events in its own way, or make use of the objects provided in the Kit with defined responses to event messages:

- The Text object responds to keyboard events by formatting and displaying the user’s typing. It responds to mouse events by altering the selection and insertion point.

- Control objects (such as Buttons, Sliders, and TextFields) capture keyboard and mouse events and turn them into action messages for other objects. By implementing a method that can respond to an action message, you can add specific behavior to your application without directly responding to the event message. (Action messages are described in a later section.)
- ScrollViews capture the mouse events that scroll one (larger) View within another (smaller) View, and do the scrolling for you.

The following sections discuss how event messages are dispensed and the Kit's default response to each one.

Keyboard Events

The Application object sends keyboard events to the key window, which passes them on as event messages to its first responder.

There's just one exception to this rule. When the Application object gets a key-down event, it checks whether the Command key was down at the time of the event. If it was, NXApp first tries to pass the event as a potential keyboard alternative—a keystroke that can activate a menu item or a button. Only after determining that no object will respond to the keyboard alternative does NXApp distribute it as an ordinary key-down event. See "Keyboard Alternatives," later in this section, for more information.

Left Mouse Events

Users can select an object on the screen (a View) by moving the cursor so that it points to the object and pressing the left mouse button. The mouse-down event is sent to the object the user selects as a **mouseDown:** event message. Mouse-dragged and mouse-up events that follow the mouse-down event are sent to the same object.

However, mouse-dragged and mouse-up events generally aren't distributed through event messages. Once an object receives a mouse-down event, its **mouseDown:** method can set up its own event loop to get these events until the user releases the mouse button. See "Modal Event Loops," later, for details.

Hit Testing

When it receives a mouse-down event, the Window object uses View's **hitTest:** method to look for the View in which the cursor was located when the mouse button was pressed—the View that contains the coordinates of the mouse-down event's **location** component. If **location** is within more than one View, **hitTest:** picks the View that's lowest in the view hierarchy.

hitTest: searches through a **subviews** list by starting at the end and working its way back toward the beginning. This gives the last subview to draw the first opportunity to accept the event. If the cursor is located in an area shared by two overlapping subviews of the same superview, as illustrated in Figure 7-3, the subview on top gets the event.

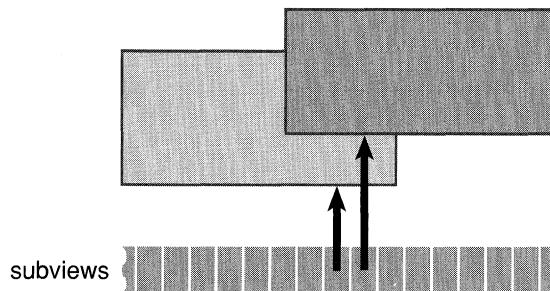


Figure 7-3. Overlapping Subviews

Trapping the First Event

A left mouse-down event selects more than just the object that's to receive subsequent mouse events. It can also select:

- The active application, if the mouse-down event spawns application-activated and -deactivated subevents.
- The key window (and main window), if the Window that receives the event isn't already the key window.
- The first responder, if the View associated with the event accepts first-responder status.

When a left mouse-down event is used to select a new key window, and possibly a new active application, you may not want it to do anything more.

Suppose, for example, that the user has two of the Quadrant windows illustrated above in Figure 7-2, “Four Text Objects in a Window,” on-screen at the same time; each of the windows has Text objects displaying editable text. By clicking first in one window, then in another, the user can repeatedly alter the key window. If the click were passed through to

the Text object each time, it would also select a new first responder and alter the current text selection or insertion point. This would make it nearly impossible to select a window without also selecting text.

To prevent this from happening, you can trap mouse-down events that choose the key window before they change the first responder and before they're sent through as event messages to Views within the Window's view hierarchy.

Whenever a Window receives a left mouse-down event that makes it the key window, it sends the View that was selected by the event an **acceptsFirstMouse** message before passing it the event. If the View returns YES, the Window will send it a **mouseDown:** message for the event. If the View returns NO, the mouse-down event won't be sent to the View, and neither will the mouse-dragged and mouse-up events that follow the mouse-down.

In general, Views that display material that the user can select or edit should return NO to an **acceptsFirstMouse** message; other Views should return YES. In the Application Kit, Text objects, TextFields, and Forms refuse the first series of left mouse events, but Sliders, Buttons, and Scrollers accept them.

The default response of the **acceptsFirstMouse** method defined in the View class is NO. To change the default in your View subclass, simply implement your own version of the method:

```
- (BOOL) acceptsFirstMouse
{
    return YES;
}
```

Selecting the First Responder

The Application Kit tries to make the View selected by a left mouse-down event the first responder. If it answers YES to an **acceptsFirstResponder** message, its Window is sent a **makeFirstResponder:** message as discussed above under “The First Responder.”

If the View selected by the mouse-down event is one that accepts being the first responder, but the current first responder refuses to give up that status (returns **nil** to a **resignFirstResponder** message), the mouse-down event and the following mouse-dragged and mouse-up events are sent to the current first responder, rather than to the View selected by the mouse-down event.

Right Mouse Events

Right mouse events within the key window are distributed very much like left mouse events. When the user presses the right mouse button while the cursor is in the key window, hit testing finds the View where the cursor is located. The right mouse-down event and subsequent right mouse-dragged and right mouse-up events are sent to that View.

However, Views rarely respond to right mouse events. In the user interface, these events are reserved for bringing a copy of the main menu to the cursor. Therefore, **rightMouseDown:** event messages generally are passed up the responder chain from next responder to next responder until they reach the Window, which then makes sure that the main menu gets the event. The main menu sets up a modal event loop that collects all subsequent right mouse events until the mouse button is released. (Event loops are discussed under “Modal Event Loops” later in this chapter, and passing event messages up the responder chain is discussed under “Event Messages in the Responder Chain” below.)

This pattern of distribution permits a View that needs to distinguish between left and right mouse events to get both. At the same time, it ensures that right mouse events that aren’t intercepted by a View do the job that they’re meant to do in the user interface.

Unlike left mouse events, right mouse events don’t change the active application, key window, or first responder, and they aren’t trapped if the View returns NO to an **acceptsFirstMouse** message.

Only Views in the key window are given a chance to respond to right mouse events. If the Window that receives a right mouse event isn’t the key window, it turns the event over to the main menu. This restriction makes the user interface for right mouse events match that of left mouse events more exactly. When the user begins working within a window using the left mouse button, the window becomes the key window. To ensure that the user’s work with the right mouse button also takes place in the key window, the window has to be the key window before any right mouse events can be distributed to its Views.

Mouse-Exited and Mouse-Entered Events

An application can receive mouse-exited and mouse-entered events, provided:

- At least one of its Windows has an event mask that accepts the events, and
- The application has set a tracking rectangle within the Window.

Mouse-exited events are generated when the cursor leaves the tracking rectangle; mouse-entered events occur when the cursor enters the rectangle.

A Window can have any number of tracking rectangles. So that particular events can be matched to particular rectangles, you can assign each rectangle an identifying tag that will be reported back in the event records of the mouse-exited and mouse-entered events that it generates.

Window's **setTrackingRect:inside:owner:tag:left:right:** method sets a tracking rectangle:

```
NXRect rect;

[aView getFrame:&rect];
[[aView superview] convertRect:&rect toView:nil];
[myWindow setTrackingRect:&rect
    inside:YES
    owner:aView
    tag:3
    left:YES
    right:NO];
```

In this example, **aView**'s frame rectangle is made the tracking rectangle, and **aView** itself is made the tracking rectangle's owner, the object responsible for handling the events the rectangle generates. The owner need not be a View, but it should be a Responder. When the Window receives mouse-exited and mouse-entered events, it dispatches **mouseExited:** and **mouseEntered:** event messages directly to the owner.

The **convertRect:toView:** message above transforms **aView**'s frame rectangle to the correct coordinate system for making it a tracking rectangle. A tracking rectangle is specified in the base coordinate system for the window—here in **myWindow**'s base coordinate system. Since a View records its **frame** instance variable in its superview's coordinate system, the message is sent to **aView**'s superview. See “Converting Coordinates” later in the chapter for more on **convertRect:toView:** and similar methods.

The rectangle's tag distinguishes it from other tracking rectangles within the same window. The inside flag indicates whether the cursor starts out inside the rectangle (YES) or outside it (NO). If it's YES, the first event received for the rectangle will be a mouse-exited event, regardless of where the cursor is actually located. If NO, the first event will be a mouse-entered event.

The last two arguments to **setTrackingRect:inside:owner:tag:left:right:** specify whether events are to be generated for the rectangle only if one or both of the mouse buttons is being held down. In this example, mouse-exited and mouse-entered events will be generated only while the left mouse button is down. This makes these events somewhat akin to left mouse-dragged events.

A tracking rectangle remains in effect until another rectangle with the same tag is set for the Window, or until it's removed by the **discardTrackingRect:** method:

```
[myWindow discardTrackingRect:3];
```

Note: Don't assign negative tags to tracking rectangles. The Application Kit uses negative numbers to identify the tracking rectangles that generate cursor-update events.

Kit-Defined Events

The application-activated and application-deactivated subevents of the kit-defined event don't generate event messages. However, they do initiate **becomeActiveApp** and **resignActiveApp** messages to the Application object, and **appDidBecomeActive:** and **appDidResignActive:** messages to the Application object's delegate, as discussed under "The Active Application" above. You should respond to these subevents only by writing methods that can respond to these messages.

The subevents of the kit-defined event that concern the state of a window generate event messages to the Window object whose window number is listed in the event record. The Window class defines methods to respond to these messages.

Window-Moved

The **windowMoved:** method updates the Window's **frame** instance variable to record the new location of the window. It then informs the Window's delegate of the move by sending it a **windowDidMove:** message, if the delegate has a method that can respond. The delegate can use Window's **getFrame:** method to get the window's new location in screen coordinates:

```
- windowDidMove:sender
{
    NXRect rect;
    [sender getFrame:&rect];
    .
}
```

The location of a window is important to the Application Kit as it responds to user actions that manipulate windows, but it's generally of little use to an application. Most applications shouldn't care where the user places windows and won't need to do anything special when a window moves.

Window-Exposed

The **windowExposed:** method redisplays part (sometimes all) of the Window's contents. The area that's redisplayed is a rectangle calculated from the event record. The location of the rectangle is taken from the event record's **location** component and its size is taken from the **data.compound.misc.L** component.

After sending a display message, **windowExposed:** informs the Window's delegate with a **windowDidExpose:** message, if the delegate has a method that can respond.

Window-Resized

The **windowResized:** method redisplays the Views within a window in response to a window-resized subevent. However, this method isn't currently used. When the user resizes a window by dragging its resize bar, no window-resized subevents are generated. Instead, the Window's frame view, which contains the resize bar, resizes the Window and redisplays its Views. Applications are informed of the resizing, not by an event, but by notification messages sent to the Window's delegate (or, in the absence of a delegate, to the Window itself):

- As the user drags an outline of the window, repeated **windowWillResize:toSize:** messages are sent to the delegate, if the delegate implements a **windowWillResize:toSize:** method. These messages give the delegate a chance to determine the new size of the window. The first argument to **windowWillResize:toSize:** is the **id** of the Window object. The second argument passes a pointer to an NXSize structure containing the proposed new width and height of the window. The delegate can alter these values to constrain the size of the window. It can be kept within maximum and minimum size limits, or be made to grow and shrink by defined amounts (as is the Workspace Manager's Directory Browser). The on-screen outline will reflect the altered values the delegate places in the NXSize structure.
- After the user releases the mouse button and just before the window is redisplayed in its new size, the delegate is informed with a **windowDidResize:** message, if it has a method that can respond. The argument passed in a **windowDidResize:** message is the **id** of the Window object, which can provide the new dimensions of the window in response to a **getFrame:** message.

If a Window doesn't have a delegate, or its delegate doesn't respond to **windowWillResize:toSize:** and **windowDidResize:** messages, these messages will be sent to the Window instead—but only if the Window can respond. This means that you can implement **windowWillResize:toSize:** and **windowDidResize:** methods either in a Window delegate or in a Window subclass.

Application-Defined Events

If your application makes use of application-defined events, you must write the code to respond to them.

When it receives the event, the Application object sends itself an **applicationDefined:** event message. But its **applicationDefined:** method does nothing more than pass the same message on to its delegate, if the delegate can respond.

You can implement an **applicationDefined:** method either in a subclass of the Application class, or in a class definition for the Application object's delegate.

System-Defined Events

Most subevents of the system-defined event are handled internally by the Application Kit. Only one, the power-off subevent, results in event messages.

Power-off subevents are generated when the user presses the Power key on the keyboard. The Window Server broadcasts the event to every application with on-screen windows.

One of the applications that receives the event is the Workspace Manager. It puts up a panel that requires users to confirm the power-off instruction or cancel it. If the user doesn't rescind the instruction, the Workspace Manager sends each application it launched a **powerOffIn:andSave:** message. This is the same message it sends to its applications when the user wants to log out.

The **powerOffIn:andSave:** message is received by the Application object, which terminates the main event loop and sends its delegate an **appPowerOffIn:andSave:** message, if the delegate has a method that can respond. The first argument to both methods is the number of milliseconds before the power goes off. The second argument is currently meaningless and should be ignored.

The Application object's delegate can ask for more time by sending the Workspace Manager an **extendPowerOffBy:actual:** message. It should then save files and take whatever other steps are necessary to prepare for the shutdown.

Applications that are launched in the workspace ignore the power-off subevent and attend only to the messages received from the Workspace Manager. Applications that are launched from the command line won't get these messages and must respond to **powerOff:** event messages instead.

NXApp is the object that receives the **powerOff:** message. If the application was launched by the Workspace Manager, its **powerOff:** method does nothing. Otherwise it sends its delegate an **appPowerOff:** message, if the delegate can respond.

Cursor Coordinates

Methods that respond to mouse events may want to note the exact location of the cursor on-screen. The coordinates in the **location** component of the event record are given in the window's base coordinate system; they can be translated to the receiving View's local coordinates by the **convertPoint:fromView:** method.

```
NXPoint where;

where = eventPtr->location;
[self convertPoint:&where fromView:nil];
```

This method transforms a point expressed in the coordinate system of its second argument to the receiving View’s reference coordinate system. When the second argument is **nil**, as it is here, it’s assumed that the point is expressed in the base coordinate system. **convertPoint:fromView:** alters the NXPoint structure referred to by its first argument and returns a pointer to the same structure.

Note that once the **location** component of the event record has been transformed from the base coordinate system, other objects in the view hierarchy will be unable to rely on it. For this reason, it’s first assigned to the local variable **where** before being passed to **convertPoint:fromView:** in the example above.

Querying the Cursor

If the process of responding to an event takes some time and you need a more recent indication of the cursor’s location, Window’s **getMouseLocation:** method provides one:

```
NXPoint where;
[myWindow getMouseLocation:&where];
```

This method places the current cursor location in the NXPoint structure referred to by its argument; it returns **self**. The point is specified in the receiving Window’s base coordinate system and can be altered by the **convertPoint:fromView:** method illustrated above.

Testing the Cursor’s Location

Methods that respond to events often must test whether the cursor is located in a particular View or in a particular region of a View, usually expressed as a rectangle. The function that makes the test is **NXMouseInRect()**.

```
BOOL inside;
NXRect rect;

[myView getBounds:&rect];
inside = NXMouseInRect(&where, &rect, NO);
```

This function takes a pointer to the cursor location and a pointer to a rectangle, here **myView**’s bounds rectangle, and returns whether the point is inside the rectangle. The third argument, **NO** in the example, is best explained by examining what’s meant by the “location of the cursor.”

The cursor’s location, as reported in the event record or by **getMouseLocation:**, is, in fact, the location of just one point on the cursor, its *hot spot*. Since the cursor is an image, displayed with pixels, the hot spot corresponds visually to a particular pixel—in effect a “hot pixel.” The pixel chosen by the hot spot is the one cursor pixel that users can’t drag off-screen.

When testing whether the cursor is inside a rectangle, what we really want to know is whether the pixel corresponding to the hot spot is inside the rectangle.

The hot spot has no fractional coordinates, so it always lies on a corner where four pixels meet. On the MegaPixel Display, the pixel chosen by the hot spot is, in accord with the rules described under “Imaging Conventions” in Chapter 4, “Drawing,” the one that lies below it and to its right. In Figure 7-4, small arrows point to the potential hot spots that would choose the “hot pixels” labeled “A” and “B”.

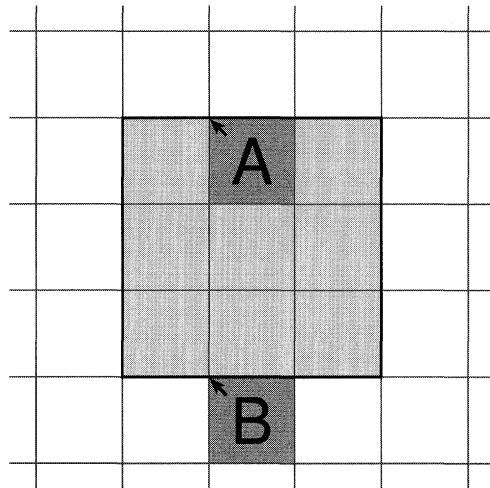


Figure 7-4. Testing the Cursor’s Location

In this diagram, pixel A lies inside the shaded rectangle and pixel B lies outside it, but both hot spots lie on the edge of the rectangle. To correctly determine which hot spot chooses a pixel inside the rectangle and which does not, `NXMouseInRect()` must know the polarity of the y-axis. When passed NO as the third argument, it assumes that y coordinate values increase from bottom to top, so hot spots with y coordinate values that match the minimum rectangle values are excluded from the rectangle. When passed YES, it assumes that the y-axis has been flipped, with coordinate values increasing from top to bottom, so hot spots with y coordinates equal to the maximum rectangle values are the ones excluded.

Flipped coordinate systems are described under “View Coordinate Systems” later in this chapter.

Event Messages in the Responder Chain

Every Window has its own responder chain of Views. When a View receives a keyboard or mouse event message that it can't handle, the message is passed to its next responder.

The chain is established as each View object has its **nextResponder** instance variable initialized to another object. As a default, and with the sole exception of the content view, a View's next responder is its superview. The content view's next responder is the Window. The Window ends the responder chain; its next responder is **nil**.

You can add other Responders into this default chain:

```
[anotherResponder setNextResponder:[self nextResponder]];
[self setNextResponder:anotherResponder];
```

The mechanism for passing events along the chain is inherited from the Responder class. Responder has a default implementation for the methods that respond to keyboard and mouse event messages. Its methods don't take any action in response to the event; they simply send the same message on to the next responder:

```
- keyDown:(NXEvent *)theEvent
{
    [nextResponder keyDown:theEvent];
}
```

For an object to actually do anything with key-down events, it must be provided with a **keyDown:** method that overrides the Responder version.

For example, suppose that a mouse-down event makes **redView** the first responder. The application then receives a key-down event that gets passed to **redView** in the form of a **keyDown:** message. However, **redView** isn't equipped to handle a key-down event; it doesn't have access to a **keyDown:** method that overrides the method defined in the Responder class. The Responder method passes the event—again as a **keyDown:** message—to **redView**'s next responder, its superview.

Figure 7-5 illustrates how the chain works. It shows part of the inheritance hierarchy for **redView** and two other objects, **greenView** and **blueView**. When **redView** receives a **keyDown:** message, it applies the version of this method it inherits from the Responder class. As shown above, Responder's **keyDown:** method simply passes the message on to **redView**'s next responder, **greenView**. **greenView** also applies Responder's version of **keyDown:**, passing the message on to its next responder, **blueView**. **blueView**'s class defines a **keyDown:** method that overrides Responder's default. It applies this method and breaks the chain.

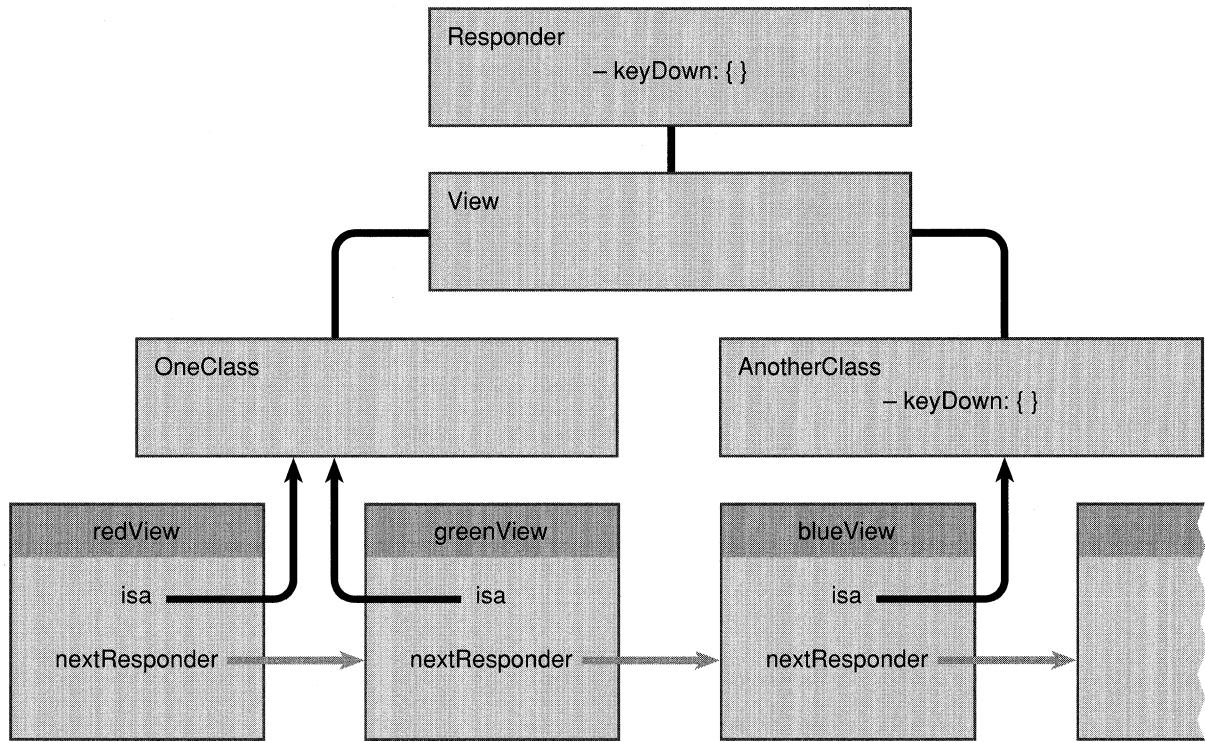


Figure 7-5. Responder Chain

By repeated iterations of the Responder method, an event message can be passed up the view hierarchy to the content view and Window. If it's passed all the way up the chain, and no object responds with a method that overrides Responder's default, the event won't be handled. It's the application's responsibility to respond to all the mouse and keyboard events it asks for.

Action Messages

Control objects give content to the user's mouse and keyboard actions. They translate the event messages they receive into more precise, application-specific action messages for other objects. A Control can be viewed as simply a tool that permits the user to give instructions to the application, a device that stands between the user and the object that will ultimately respond to the user's event.

The Control classes defined in the Application Kit—Button, Scroller, Slider, TextField, Matrix, and Form—are described in Chapter 9 under “Controls.” Control itself is an abstract superclass; it defines a paradigm for inter-object communication—action messages—that its subclasses inherit and other objects emulate. You can define your own Control subclasses to take advantage of this paradigm.

Controls use ActionCell objects of various sorts to hold information about their internal state. The ActionCell superclass defines instance variables for the two elements essential to an action message:

target The object that's responsible for responding to the user's action on the Control

action The method that specifies what the target is to do

The Control class defines an instance variable that can identify the sender of an action message:

tag A number that the target can use to distinguish among Controls that send the same action messages

Each ActionCell can also have its own tag.

A Control can send a different action message to a different target for each ActionCell it contains. Matrix and Form objects typically contain more than one ActionCell, but Buttons, Sliders, and TextFields are single-Celled. The content area of a Menu is filled by a Matrix of ActionCells. Scrollers aren't composed of Cells; they define their own **target**, **action**, and **tag** instance variables.

The Target

The target is the object that receives the action message. Setting a target

```
[myControl setTarget:messageHandler];
[myMatrix setTarget:messageHandler at:rowSix :colZero];
```

ties the Control to a specific object that's expected to respond to all its action messages; no other object will ever have the opportunity to respond. Typically, the target is another View in the same Window as the Control, the Window's delegate, or an object of your own design.

The **target** method returns the Control's current target:

```
id bullseye;
bullseye = [myControl target];
```

Some Controls can't be tied to a particular target. The Cut command in the Edit menu, for example, can delete material first in one window, then in another. The user selects the receiver of the Cut command's action message by picking the key window (and main window) and choosing the Window's first responder. The command deletes the selection owned by the first responder in the key or main window.

Controls like this have their targets set to **nil**. The mechanism for dispatching action messages finds an appropriate receiver for the message each time the message is sent. This mechanism is explained under “Action Messages in the Responder Chain” below.

The Action

The **action** instance variable names the method the target is asked to perform. It’s a method selector, assigned with the Objective-C `@selector()` operator:

```
[myControl setAction:@selector(doBehave:)];
[myMatrix setAction:@selector(doBeNice:) at:rowOne :colThree];
```

Action methods take a single argument, the **id** of the Control object that sends the message. This argument enables the receiver to ask the Control for more information, if it’s needed. For example, a target receiving an action message from a Button might want to learn the Button’s current state:

```
- doReact:sender
{
    int setting;
    setting = [sender state];
    .
}
```

Each Control dispatches action messages in response to a different set of user events:

- A Button generally sends action messages on a mouse-up event, if it also received the mouse-down event and the cursor is inside its frame rectangle when the mouse button goes up.
- A repeating Button sends action messages continuously, as long as the user holds the mouse button down and keeps the cursor inside its frame rectangle.
- A Slider also can send action messages continuously, as long as the mouse button is held down.
- A Slider that doesn’t send continuous action messages can send them on a mouse-down event, on a mouse-up event, or for each mouse-dragged event that repositions its knob.
- A TextField sends action messages when the user presses Return after entering data in the field, but only if the data is acceptable.

Although the `ActionCell` class provides for only one action selector, `Control` (and `Cell`) subclasses can define any number of other actions for special circumstances, and dispatch them when they want to. For example, a Control could send one message when it’s clicked and another when it’s double-clicked.

Some Controls may want to give their targets an opportunity to take action when the mouse button first goes down, and so may dispatch an action message on a mouse-down event. For example, a Button that caused its target to change shape might want the target to assume its altered shape, temporarily, on a mouse-down event so that the user can see what the Button does. The change would become permanent only on a mouse-up event.

The method that Controls use to send action messages is **sendAction:to:**. It takes the action selector and target object, which can be **nil**, as arguments.

```
[myControl sendAction:@selector(doReact:) to:messageHandler];
```

If the selector for the action message is **NULL**, no message is sent.

Since both the target and the action are passed as parameters to **sendAction:to:**, Control subclasses are free to define multiple target objects and action selectors for special circumstances.

Action Messages in the Responder Chain

When the target of an action message is **nil**, the Control that's about to send the message must look for an appropriate receiver. It conducts its search in a prescribed order:

- It begins with the first responder in the current key window and follows **nextResponder** links up the responder chain to the Window object. After the Window object, it tries the Window's delegate.
- If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the Window object and its delegate.
- Next, it tries the Application object, NXApp, and finally the Application object's delegate. NXApp and its delegate are the receivers of last resort.

This search path is illustrated below in Figure 7-6.

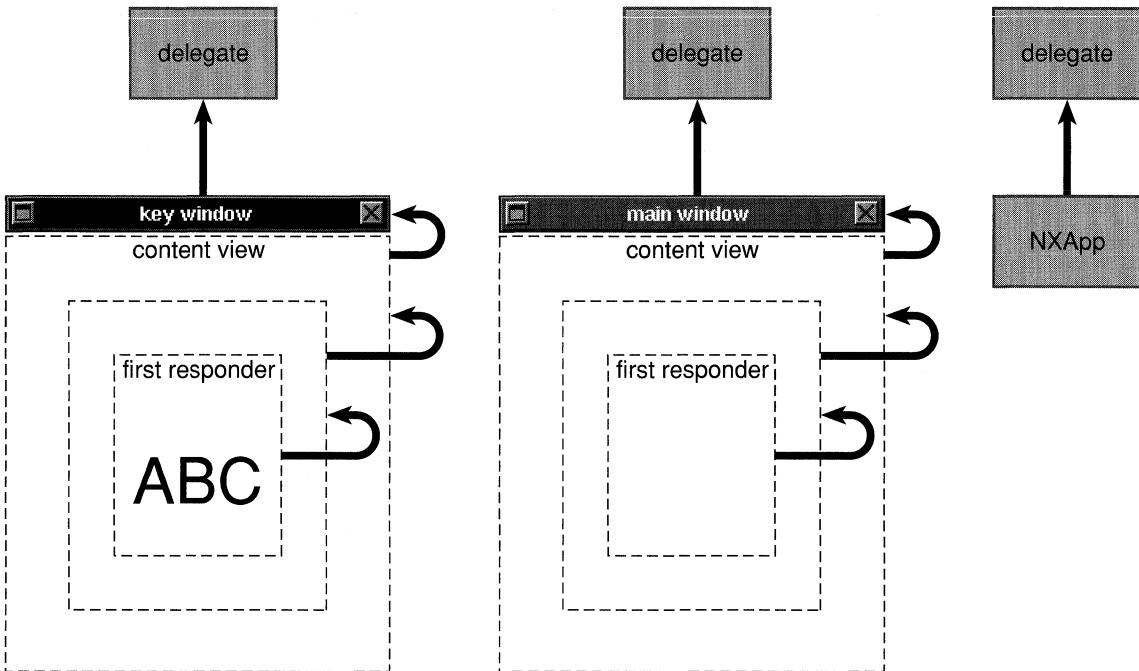


Figure 7-6. Search Path

The search stops as soon as an appropriate receiver is found for the action message. Most often, the first responder in the key or main window will handle it.

To end the search for a receiver, two things are required:

- An object must have a method that can receive the action message.
- The method must return a value other than **nil**.

If an object has access to a method matching the action selector, the message is sent. If the object then returns a value other than **nil**, the search for a receiver is aborted.

By returning **nil**, a method permits objects farther along the search path to receive the same action message. This is useful when the first responder and all the Views above it in the view hierarchy should have a chance to react to the message. However, returning **nil** isn't typical. Most methods that handle action messages should return **self** to stop the message from going any farther.

This way of finding receivers for untargeted action messages serves a variety of different messages. It locates the receiver for:

- Action messages destined for the first responder, such as those sent from the Font panel or by the Cut, Copy, and Paste commands.
- Action messages destined for the key window or main window, such as those sent by the Miniaturize and Close commands.

- Action messages destined for the Application object, such as those sent by the Hide and Quit commands.

You can take advantage of the fact that the Application object and its delegate end the search path by implementing default methods that respond to action messages in an Application subclass, or in the delegate's class.

Kit-Defined Action Methods

The Application Kit defines a number of methods that can respond to action messages; they each take a single argument, the **id** of the sender. Often the Kit-defined method doesn't do anything with the argument; it's there just so the method selector can be used in action messages.

The chart below lists, by class, some of the action methods defined in the Kit:

Application

hide:	Hides the application's windows for the Hide command.
unhide:	Unhides and activates the application.
stop:	Stops the main event loop without terminating the application.
terminate:	Terminates the application and exits for the Quit command.

Button and ButtonCell

performClick:	Simulates clicking the Button or ButtonCell.
---------------	--

Control and Cell

takeIntValueFrom:	Resets receiver's int value to that of the sender.
takeFloatValueFrom:	Resets receiver's float value to that of the sender.
takeDoubleValueFrom:	Resets receiver's double value to that of the sender.
takeStringValueFrom:	Resets receiver's string value to that of the sender.

Matrix

selectAll:	Selects all the Cells in the Matrix.
selectText:	Selects all the text in the first editable Cell of the Matrix (or in the last editable Cell if the message is the result of Shift-Tab).

PopUpList

popUp:	Puts the PopUpList on-screen under the cursor.
--------	--

TextField

selectText:	Selects all the text in the TextField object.
-------------	---

Text

selectAll:	Makes the Text object the first responder and selects all its text.
selectText:	Makes the Text object the first responder and selects all its text, just as selectAll: does.
copy:	Copies selected text to the pasteboard for the Copy command.
paste:	Retrieves text from the pasteboard and replaces the current selection with it, for the Paste command.
delete:	Deletes selected text (without putting it in the pasteboard) for the Delete command.
cut:	Copies selected text to the pasteboard and deletes it for the Cut command.
changeFont:	Dispatches a convertFont: message to the sender, allowing the sender to convert the font of the current selection (or of all the text if the Text object doesn't support multiple fonts).
superscript:	Raises the current selection above the baseline.
subscript:	Lowers the current selection below the baseline.
unscript:	Returns the current selection to the baseline.

View

printPSCode: Prints the View, including all its subviews.

Window

miniaturize: Miniaturizes the Window.

demiiniaturize: Restores a miniaturized window to the screen.

orderFront: Places the Window at the front of its tier on-screen.

orderBack: Puts the Window at the back of its tier on-screen, but in front of the workspace window.

orderOut: Removes the Window from the screen.

makeKeyAndOrderFront: Makes the Window the key window and puts it on-screen at the front of its tier.

performClose: Simulates clicking the close button for the Close command.

performMiniaturize: Simulates clicking the miniaturize button for the Miniaturize command.

printPSCode: Prints all the Views within the Window (including the frame view).

smartPrintPSCode: Prints all the Views within the Window (including the frame view) and tries to intelligently format the pages.

The Tag

Each Control can be assigned an arbitrary integer as a tag.

```
[myControl setTag:34];
```

Each ActionCell in a multi-Celled Control can also be assigned a tag:

```
[myMatrix setTag:14 at:rowOne :colFour];
```

The tag is a convenience for identifying the Control or Cell in the code you write. If a number of different Controls have the same target and the same action selector, the target can use the tag to distinguish among the Controls:

```
- doReact:sender
{
    switch( [sender tag] ) {
        case 0:
            .
            .
            .
            break;
        case 1:
            .
            .
            .
            break;
        case 2:
            .
            .
            .
            break;
        default:
            break;
    }
}
```

The Sender as an Argument

The method for dispatching action messages, `sendAction:to:`, is defined in the Control class and is therefore available only to Control subclasses.

Nevertheless, one crucial element of the action-message paradigm is emulated throughout the Application Kit and in many application programs: Many objects include their own **ids** in the messages they send. This serves to reduce the number of arguments that the message requires. Instead of passing every possible value that any receiver might need, each receiver can send messages back to the sender to get just the values it requires.

For example, when a Matrix object tells one of its cells to draw itself, it informs the cell which Matrix sent the message:

```
[aCell drawSelf:&rect inView:self];
```

The first argument specifies the rectangle where the Cell is to draw; any other information the Cell may need, it can get from the Matrix.

If you define a method that requires information from the sender, it's a good idea to make the sender's **id** be the method's only argument. This both simplifies the calling sequence and makes the method's selector eligible to be used in action messages.

If you define a method that requires no arguments, it sometimes makes sense to add an object **id** as an argument anyway. Even though the method won't look at the **id** it's passed, it too will be eligible to respond to action messages.

Keyboard Alternatives

Most graphic objects respond when the user both presses and releases the mouse button as the cursor points to the object. Objects can be programmed to respond in an identical manner to a single keystroke (usually modified by the Command key), no matter where the mouse is pointing. By treating the keystroke as equivalent to a mouse click, they provide users with a *keyboard alternative* to the mouse.

From Key Down to Key Equivalent

Keyboard alternatives are implemented by translating a key-down event into a **performKeyEquivalent:** message. **performKeyEquivalent:** takes a single argument, a pointer to the event record of the key-down event. To implement a keyboard alternative, an object must look into the record to see which character was typed. If the character corresponds to the character cached by the object as its *key equivalent*, it returns YES; otherwise, it returns NO. For example:

```
- (BOOL)performKeyEquivalent:(NXEvent *)theEvent
{
    if ( theEvent->data.key.charCodeAt == myKeyEquivalent ) {
        [self doWhatAMouseClickWouldDo];
        return( YES );
    }
    return( NO );
}
```

This method identifies the key equivalent by the character code in the event record; it ignores the character set and key code. The key equivalent is the character that was typed, not the key that was pressed. Uppercase “F” and lowercase “f” are different key equivalents, and both are distinct from Control-F. If the user changes the way keys are mapped to character codes, the key that must be pressed to activate a keyboard alternative may also change. The default keyboard mapping is given in the *NeXT Technical Summaries* manual.

The Button class implements a **performKeyEquivalent:** method much like the one shown above, so Buttons have a built-in ability to respond to keyboard alternatives. The Matrix class implements a more elaborate version that gives each ButtonCell (including each MenuCell) the opportunity to have its own key equivalent. You need only name the character. **setKeyEquivalent:** makes the assignment and **keyEquivalent** returns the current key equivalent:

```
unsigned short theKey;

[myButton setKeyEquivalent:'f'];
theKey = [myButton keyEquivalent];
```

By default, all other event-handling objects (all Responders) return NO when sent a **performKeyEquivalent:** message. To have one of these objects respond to a key equivalent, you must override the default with a **performKeyEquivalent:** method of your own, like the one illustrated above.

Making sure that an object can respond to the appropriate keystroke (by setting its key equivalent or providing it with a **performKeyEquivalent:** method) is only the first step. You may also have to take some steps to ensure that **performKeyEquivalent:** messages reach the object.

Command Key-Down Events

The Application Kit interprets every key-down event as a potential keyboard alternative, if the Command key was pressed at the time of the event. It distributes the event in a way that's designed to initiate **performKeyEquivalent:** messages to Views that may have key equivalents.

The first step in this process is for the Application object to pass the event, in the form of a **commandKey:** message, to the Windows in its window list.

The second step is for a Window to translate the **commandKey:** message into a **performKeyEquivalent:** message to its Views.

The third step is to pass the **performKeyEquivalent:** message down the view hierarchy until it reaches the View that will respond.

commandKey: Messages

In the main event loop, the Application object checks every key-down event it receives from the Window Server to see whether the Command key flag is set. If it is, instead of dispatching the event like other key-down events, NXApp sends a **commandKey:** message to each Window in its window list, until one of the Windows returns YES. By default, a Window object does nothing more than return NO, causing NXApp to send the message to the next Window in the list.

In this way, a Command key-down event is distributed to all the Windows in the application, not just to those that would normally respond to keyboard events, and not just to those that are in the screen list. Hidden and miniaturized Windows also get the message.

Initiating performKeyEquivalent: Messages

By simply returning NO, a Window effectively prevents the objects within its view hierarchy from responding to the event. If there are Views within the Window that might want to handle the key equivalent, its **commandKey:** method must give them a chance to respond before returning.

The Panel class implements a version of **commandKey:** that translates the message to a **performKeyEquivalent:** message for its contentView:

```
- (BOOL) commandKey : (NXEvent *)theEvent
{
    return [contentView performKeyEquivalent:theEvent];
}
```

This method returns the same value that's returned to it by **performKeyEquivalent:** If **performKeyEquivalent:** returns YES, **commandKey:** returns YES to terminate the search for an object that can handle the key equivalent. If **performKeyEquivalent:** returns NO, **commandKey:** also returns NO, and the **commandKey:** message is passed to the next Window in the window list.

Passing performKeyEquivalent: Messages

View objects inherit a version of **performKeyEquivalent:** that recursively passes the message down the view hierarchy from subview to subview, stopping only when a subview accepts the key equivalent and returns YES. This default doesn't enable a View to respond to a key equivalent, but it lets the message reach the lowest branches of the view hierarchy, where the Control objects most likely to respond are located.

If a Window knows exactly which of its Views might respond to the Command key-down event, it can send the **performKeyEquivalent:** message directly to them. If not, it can send the message to its contentView and rely on View's default method to pass the message down the view hierarchy. The Panel method illustrated above does just that.

Unhandled Messages

Through the combination of **commandKey:** and **performKeyEquivalent:** messages described above, a Command key-down event can make its way down the entire window list in search of a View that can respond to the character that was typed. Once that View is found, YES is returned up the chain to the Application object, which originated the **commandKey:** message. Since a Responder has been found for the key-down event, the Application object does nothing more.

However, if no View is found to handle the Command key-down event, the Application object receives NO in return. Since no object has yet responded to the event, NXApp dispatches it like any other key-down event; it sends the event to the key window, which dispatches a **keyDown:** message to its first responder.

View Methods

Any View can originate **performKeyEquivalent:** messages based on the ordinary key-down events it receives. Key-down events are normally passed to the key window's first responder as **keyDown:** messages. To handle the event as a keyboard alternative, the first responder must translate it to a **performKeyEquivalent:** message:

```
- keyDown: (NXEvent *)theEvent
{
    if ( theEvent->flags & NX_COMMANDMASK )
        [self performKeyEquivalent:theEvent];
    return( self );
}
```

In this example, the first responder doesn't do anything with key-down events except treat them as potential keyboard alternatives. It checks to be sure the Command key was pressed at the time of the event and, if it was, initiates the **performKeyEquivalent:** message.

You may sometimes want objects to respond to keyboard alternatives without the Command key. For example, if you display a graphic keypad on the screen, with a numbered Button for each of the keys, you'd probably want the user to be able to simply press the numbered keys on the keyboard to operate the keypad. You probably wouldn't want to force the user to press the Command key to get the key equivalents to work. In this case the **keyDown:** method can be simplified:

```
- keyDown: (NXEvent *)theEvent
{
    [self performKeyEquivalent:theEvent];
}
```

Assuming that the first responder in the example above doesn't implement its own version of **performKeyEquivalent:**, the View version would pass the **performKeyEquivalent:** message to its first subview, then to each of the subview's subviews, then on to their subviews. If none of them recognized the key equivalent as its own, the message would be passed to the first responder's second subview, and so on, until a View responded YES or the subviews list was exhausted.

If an object has a key equivalent and it also has subviews with their own key equivalents, its version of **performKeyEquivalent:** should perform the View method so that its subviews can have a chance to respond in case it can't:

```
- (BOOL)performKeyEquivalent:(NXEvent *)theEvent
{
    if ( theEvent->data.key.charCodeAt == myKeyEquivalent ) {
        [self doWhatAMouseClickWouldDo];
        return( YES );
    }
    return( [super performKeyEquivalent:theEvent] );
}
```

Changing the Cursor

The cursor, usually a small diagonal arrow, sometimes changes to another image. It changes for two very different reasons:

- When the cursor is over a particular area of the key window, it may change to indicate the type of operation that's permitted within that area. For example, the cursor changes to an I-beam when it's over selectable text in the key window.
- When the active application is busy and unable to accept events, the cursor may change to the "wait cursor" image.

This section explains how to use the Application Kit to change the cursor image.

Defining a Cursor

A Cursor is a kind of Bitmap object. For the MegaPixel Display, it should be a bitmap 16 pixels wide by 16 pixels high. Cursors are generally opaque images surrounded by transparent pixels. When it's placed on-screen (by a Sover compositing operation), you see only the irregular opaque shape.

The Application Kit provides three ready-made Cursor objects for commonly used cursor images:

- NXArrow is the standard arrow cursor.
- NXIBeam is the I-beam used for selectable text.
- NXWait is the "wait cursor" that indicates that the application is busy.

You can create your own Cursor object just as you would any other Bitmap. The example below creates an "X" cursor and puts the hot spot (the point that's aligned with the mouse) where the two lines cross.

First a function that draws a small “X” on a transparent background is defined:

```
defineps drawX()
    0 0 16 16 Clear compositerect
    1 setalpha
    0 setgray
    1 setlinewidth
    newpath
    2 2 moveto 15 15 lineto stroke
    15 2 moveto 2 15 lineto stroke
endps
```

Next, the function is used to create the Cursor bitmap:

```
id myImage;

myImage =[Cursor newSize:16.0 :16.0 type:NX_ALPHABITMAP];
[myImage lockFocus];
drawX();
[myImage unlockFocus];
```

Finally, the hot spot is set at the upper left corner of the pixel at the center of the image. By default, Cursors are drawn in a flipped coordinate system, with y coordinate values increasing from top to bottom, so that point is (8.0, 8.0):

```
NXPoint mySpot;

mySpot.x = mySpot.y = 8.0;
[myImage setHotSpot:&mySpot];
```

Cursors can also be created from TIFF files, by imaging bitmap data, or by using any of the other variety of methods provided in the Bitmap class.

Cursor Rectangles

If you associate a cursor image with a rectangular area within a window, the Application Kit will change the cursor to that image whenever the window is the key window and the cursor is over the rectangle. For example, if you provide a little scratch pad in a window where the user can doodle, you could change the cursor to a pencil when it's over the pad.

The actual work of changing the cursor is handled by the Kit. Your application need only define a *cursor rectangle* and associate it with a particular Cursor object. When the cursor passes into the rectangle, it will be changed to the image defined by the object. When it moves out of the rectangle, it will revert to the previous image.

Cursor rectangles are implemented as specially marked tracking rectangles. Mouse-entered and mouse-exited events for the rectangles are translated into cursor-update events. The Kit receives these events and changes the cursor as required; your application should never need to look at them.

Note: Because cursor rectangles are based on tracking rectangles, a window must permit mouse-entered and mouse-exited events in its event mask for automatic cursor updating to work. It doesn't matter whether or not the mask also contains cursor-update events.

A window can have more than one cursor rectangle, but if any two intersect, one of them must be totally inside the other. Cursor rectangles can be nested or they can cover independent areas within the window, but they can't partially overlap.

Cursor rectangles exist only when the window is the key window. When a window ceases to be the key window, its cursor rectangles are discarded. When it becomes the key window again, they're reestablished. The Window object keeps a record of all its cursor rectangles so that it can discard them and reestablish them again—without any help from the application—as it loses and regains key window status.

Registering and Resetting Cursor Rectangles

Although cursor rectangles are recorded and maintained by Windows, they're established by Views. Often a View uses its own frame rectangle as a cursor rectangle, or perhaps the frame rectangles of its subviews or Cells.

A Window's set of cursor rectangles is therefore apt to change as its Views change. Any number of circumstances can invalidate a Window's cursor rectangles, including these:

- A View is resized.
- The window is resized, causing the Views inside it to be resized or to adjust their locations.
- The contents of a View are scrolled.
- The Window's view hierarchy is rearranged.

For example, if a View is removed from the view hierarchy, some cursor rectangles might need to be removed with it. Scrolling a View that contains a cursor rectangle could relocate the rectangle. If the rectangle is scrolled completely out of view, it should be removed from the Window's record of cursor rectangles. If it's partly in view and partly out, it should be updated to reflect only the part that's visible. If the rectangle stays in view, but shifts its position, the Window should record its new location.

It's therefore not enough for a View to register its tracking rectangles just once. It must be prepared to reregister them whenever the need arises.

Before getting an event in the main event loop (or in a modal loop for an attention panel), the Application Kit checks whether the key window's cursor rectangles remain valid. Any of the circumstances listed above might invalidate them. They can also be explicitly invalidated by the application:

```
[myWindow invalidateCursorRectsForView:myView];
```

If any rectangles are invalid, they're discarded and **resetCursorRects** messages are initiated to reregister the correct rectangles. Each View should define its own version of the **resetCursorRects** method. Because a new Window is flagged as having invalid cursor rectangles, this method will not only respond to periodic **resetCursorRects** messages as the application runs, it will also set up the View's cursor rectangles in the first place. To make automatic cursor updating work for your application, all you need to do is implement a **resetCursorRects** method for any View you want associated with a special cursor.

A **resetCursorRects** method should contain one or more **addCursorRect:cursor:** messages to register the View's cursor rectangles with the Window. Each message associates a Cursor object with a particular rectangle. In the example below, **resetCursorRects** sets up a cursor rectangle equal to the receiving View's bounds rectangle and associates it with the "X" cursor defined above.

```
- resetCursorRects
{
    [self addCursorRect:&bounds cursor:myImage];
    return self;
}
```

The Matrix, Form, and TextField classes use this mechanism to display an I-beam cursor over editable text. You don't have to register these cursor rectangles yourself.

The **removeCursorRect:cursor:** method removes a cursor rectangle from the Window's record and **discardCursorRects** removes all the cursor rectangles registered for the View.

```
[myView removeCursorRect:&bounds cursor:NXIBeam];
[otherView discardCursorRects];
```

The Window class defines a matching set of methods for adding and removing cursor rectangles. For example:

```
[myWindow addCursorRect:&someRect
                    cursor:myImage
                    forView:myView];
```

But the Window methods expect the rectangles to be specified in the base coordinate system. For this reason, the View methods are generally easier to use.

Wait Cursors

The wait cursor informs users that the active application is busy and will therefore be unresponsive. Its disappearance lets users know that the application is again ready to receive events.

When the wait cursor is removed from the screen, the cursor should revert to whatever image is appropriate for its location and the current state of the application. This is done automatically when a **push** message is used to set the wait cursor image and a **pop** message is used to remove it. These messages should bracket the code that the user must wait for:

```
[NXWait push];
/* code for a task that may take some time to execute */
[NXWait pop];
```

The wait cursor should remain on-screen for comparatively short periods of time. To inform users that the application will be busy for an extended period of time, don't rely on the wait cursor, but use an attention panel instead. See "Modal Sessions" under "Modal Event Loops" below.

Modal Event Loops

Part of an object's response to an event can be to get another event directly from the Application object:

```
NXEvent *myEvent;
myEvent = [NXApp getNextEvent:myMask];
```

This procedure short-circuits the main event loop, enabling an object to retrieve and respond to an expected event more quickly. When breaking into the main event loop, objects generally set up their own temporary, modal event loops for short periods of time.

getNextEvent: takes an argument, an event mask that limits the types of events it will return. The argument doesn't affect the events sent to the application by the Window Server—that is, it doesn't change any window's event mask. If the next event in the queue isn't one you want **getNextEvent:** to return, it skips over the event and continues checking until it finds one that matches the mask. Each time **getNextEvent:** checks for a new event, it begins at the beginning of the event queue and picks the first event matching its mask. The main event loop will pick up any skipped-over events.

When your program breaks into the main event loop to get events on its own, it enters a mode, a period of time when the user's actions are interpreted only by the object getting the events. The object getting the events usually sets up its own event loop as a subloop under the main event loop. The program stays in the mode until the user takes the required action to break out of the loop.

Setting up a modal event loop should be limited to cases where you can be fairly certain that the next event belongs to a limited set. You'd most often do it in response to a **mouseDown:**, **keyDown:**, or **flagsChanged:** message:

- When a View receives a mouse-down event, a mouse-dragged or mouse-up event is likely to be the next event of interest. A mouse-up event breaks the loop, so the mode lasts only while the mouse button is held down.

- When a View receives a key-down event and there's reason to believe the user has started typing, a key-up event or another key-down event is likely to follow. Any other event type serves to break the loop and end typing mode, so it won't appear to be a mode at all from the user's point of view.
- When the application receives a flags-changed event indicating that the user has pressed a modifier key, it can enter a mode until another flags-changed event indicates that the key has been released.

Since the Text, TextField, and Form objects can handle the typing needs of most applications, you generally won't need to set up a modal loop to handle keyboard events. Spring-loaded modes triggered by a modifier key are similarly rare.

Coordinating Mouse Events

Mouse-down and mouse-up events often need to be coordinated:

- Views shouldn't normally respond to a mouse-up event unless they've also received the mouse-down event that preceded it.
- Views receiving a mouse-down event should generally wait until the mouse-up event before committing themselves to an irreversible action. Users are allowed to change their minds after pressing a mouse button. If they move the cursor from the View before releasing the button, the View shouldn't respond. This is especially true of Views that respond in the control-action paradigm of the user interface.

Coordinating mouse-down and mouse-up events is easier if the mouse-up event is received as part of the response to the mouse-down event. When a **mouseDown:** method breaks into the main event loop to get the next mouse-up event, the application's response to both events is systematically integrated.

There are also reasons why mouse-dragged events are best bracketed by mouse-down and mouse-up events in a modal event loop:

- An application shouldn't ask for mouse-dragged events until it's ready to process them, and it should stop asking for them when it no longer needs them. A mouse-down event signals when it's appropriate to start receiving mouse-dragged events; the succeeding mouse-up event signals when it's appropriate to stop.
- As it waits for the mouse-up event it's interested in, an object may want to keep track of the position of the cursor, so that it can respond appropriately if the user stops pointing at the object that received the mouse-down event. For this reason, it might ask for mouse-dragged (or mouse-exited and mouse-entered) events in addition to mouse-up events.

The following example shows how a **mouseDown:** method can set up an event loop for mouse-dragged and mouse-up events.

```
- mouseDown: (NXEvent *)thisEvent
{
    register int    inside;
    int            shouldLoop = YES;
    int            oldMask;
    NXEvent        *nextEvent;

    [self doMyOwnHighlight];
    oldMask = [window addToEventMask:NX_LMOUSEDRAGGEDMASK];

    while (shouldLoop) {
        nextEvent = [NXApp getNextEvent:(NX_LMOUSEUPMASK |
                                      NX_LMOUSEDRAGGEDMASK)];

        inside = NXMouseInRect([self convertPoint:
                               &nextEvent->location
                               fromView:nil],
                               &bounds,
                               [self isFlipped]);

        switch (nextEvent->type) {
            case NX_LMOUSEUP:
                shouldLoop = NO;
                if (inside) {
                    [self doMyOwnHighlight];
                    [self doMyOwnThing];
                }
                [self doMyOwnUnhighlight];
                break;
            case NX_LMOUSEDRAGGED:
                if (inside)
                    [self doMyOwnHighlight];
                else
                    [self doMyOwnUnhighlight];
                break;
            default:
                break;
        }

    }
    [window setEventMask:oldMask];
    return(self);
}
```

This method first declares its local variables and highlights the View in response to the mouse-down event. It then resets the window's event mask to include left mouse-dragged events. (Mouse-up events are included in every window's default event mask; they shouldn't be added to the event mask at the time of the mouse-down event.) But before changing the mask, it stores the window's old event mask in a local variable (**oldMask**) so that it can be reset later.

Next, the method enters an event loop, looking only for mouse-up and mouse-dragged events. As it receives each event, the location of the mouse is translated to the View's own coordinates (by the **convertPoint:fromView:** method) and is tested against its bounds rectangle (by the **NXMouseInRect()** function) to see whether or not it lies inside the View.

The loop continues until the mouse button is released. If it's released while the mouse is pointing to the View, the View performs its **doMyOwnThing** method. In any case, the loop exits, leaving the Application object's event loop as the only one operating.

As the modal loop waits for a mouse-up event, mouse-dragged events keep track of the location of the mouse. As long as it's pointing within the View, highlighting continues. If it leaves the View, highlighting ends.

After the modal loop exits, the **mouseDown:** method resets the event mask to make sure that mouse-dragged events will no longer be sent to the Window.

This example is more an outline than an excerpt from a real program. You would need to fill in the details to turn it into a useful method for your application.

Getting and Peeking at Events

The **getNextEvent:** method mentioned above is just one of a set of five methods that you can use to read events from the event queue. All five take an event mask specifying which events they're to return, and all five skip over any events in the queue that don't match the mask.

Some of the methods "get" the event by copying it to NXApp's **currentEvent** instance variable and removing it from the queue. **getNextEvent:** is one of those methods:

```
NXEvent *eventPtr;  
eventPtr = [NXApp getNextEvent:(NX_KEYDOWNMASK | NX_KEYUPMASK)];
```

Other methods simply "peek" at the event. They leave it in the queue but copy it to memory provided by the application. Here the event is read into the **thisEvent** structure:

```
NXEvent thisEvent, *eventPtr;  
eventPtr = [NXApp peekNextEvent:NX_ALLEVENTS into:&thisEvent];
```

Peeking at an event is appropriate, for example, in modal event loops that handle key-down and key-up events. Since any other event type breaks the loop, the procedure that gets keyboard events must first peek at the next event to make sure it is, in fact, a keyboard event. If it is, the **getNextEvent:** method can be used to get it. If it's not, it should be left in the queue for another procedure to handle. Always be certain that you can respond to an event before getting it from the queue.

All five methods that read events from the queue return a pointer to the event. After the **peekNextEvent:into:** message is sent in the example above, **eventPtr** points to **thisEvent**. The pointer returned by **getNextEvent:** is the same one that **currentEvent** returns:

```
eventPtr = [NXApp currentEvent];
```

These two “get” and “peek” methods differ in one other respect: If there is no event in the queue (at least none matching the mask passed to the method), **getNextEvent:** waits for one, but **peekNextEvent:into:** doesn’t wait. It immediately returns a NULL pointer.

A third method, **peekAndGetNextEvent:**, gets events like **getNextEvent:**, but, like **peekNextEvent:into:**, returns immediately if there’s no matching event in the queue:

```
eventPtr = [NXApp peekAndGetNextEvent:NX_ALLEVENTS];
```

There are also “get” and “peek” methods that let you specify how long they should wait for the next event:

```
NXEvent thisEvent, *eventPtr;
eventPtr = [NXApp getNextEvent:NX_APPDEFINEDMASK
               waitFor:10.0
               threshold:NX_BASETHRESHOLD];
eventPtr = [NXApp peekNextEvent:NX_ALLEVENTS
               into:&thisEvent
               waitFor:10.0
               threshold:NX_BASETHRESHOLD];
```

The time is specified in seconds; in the examples above, **getNextEvent:waitFor:threshold:** and **peekNextEvent:into:waitFor:threshold:** will both wait for 10 seconds before returning NULL. If an event which matches the mask is queued before the 10 seconds are up, they’ll return with the event immediately.

The last argument to both of these methods is a priority threshold that’s explained in the next section.

Scheduling

While your application is responding to an event, it can’t be doing anything else. Between events, however, it can turn to other tasks. It could:

- Execute a timed entry, if one is due.
- Respond to a message from another application, if one has been received.
- Read data from a file descriptor, if there’s any data to read.

Whenever the application is ready to get (or peek at) another event, it can call timed entries, procedures to handle messages received at a port, and procedures to read data received at a file descriptor. To be eligible, a procedure must be registered using the appropriate method in the Display PostScript client library:

- DPSAddTimedEntry()
- DPSAddPort()
- DPSAddFD()

A procedure is called only if there's work for it to do—the timed entry must be due, a message must have been received at the port, data must be ready at the file descriptor—and only if it's scheduled.

Whether or not a procedure is scheduled depends largely on its priority level. The priority is an integer between 0 and 30, with 30 as the highest possible priority and 0 as the lowest. The priority of a procedure is set when it's first registered (with the **DPSAddTimedEntry()**, **DPSAddPort()**, and **DPSAddFD()** functions).

Whenever an application gets (or peeks at) the next event, it specifies a priority threshold. Procedures with priorities lower than the threshold are screened out; all those with equal or higher priorities are checked at least once to see whether they should be called before the “get” or “peek” method returns. If the method doesn't return immediately but waits for an event to arrive in the queue, it's possible for a procedure to be checked and called many times over. Even if the method returns without waiting, it's guaranteed that each procedure at or above the threshold will be checked once.

The Application Kit makes use of three priority thresholds and defines a constant for each:

10	NX_MODALRESPTHRESHOLD
5	NX_RUNMODALTHRESHOLD
1	NX_BASETHRESHOLD

The main event loop gets events at the lower priority threshold of 1; it's very permissive about what procedures can be called between events. But when a Control, Text object, or other Responder sets up a modal event loop to get its own events, it gets (or peeks at) them at the higher priority threshold of 10. Since its purpose is to narrow the application's focus for a short period of time, a modal loop is more restrictive than the main event loop.

Attention panels also set up modal event loops, but at the less restrictive threshold of 5. These event loops are discussed below under “Modal Windows” and “Modal Sessions.”

A Listener object registers the application's port for receiving messages at the base priority of 1; the application can respond to messages from the Workspace Manager or other applications while it's getting events in the main event loop, but not while it's in a modal event loop. On the other hand, the Text object registers a timed entry to blink the caret at a priority level of 5; the caret will blink even in an attention panel, but not while the Text object is in its modal loop getting and responding to keyboard events, and not while a Button or Slider in the same window is responding to the user's mouse events.

The **getNextEvent:**, **peekNextEvent:into:**, and **peekAndGetNextEvent:** methods discussed earlier all specify a threshold of 10; they're designed for modal event loops.

The remaining two "get" and "peek" methods allow you to specify the threshold:

```
NXEvent *eventPtr;
eventPtr = [NXApp getNextEvent:NX_ALLEVENTS
                           waitFor:NX_FOREVER
                           threshold:NX_BASETHRESHOLD];

NXEvent thisEvent, *eventPtr;
eventPtr = [NXApp peekNextEvent:NX_MOUSEUPMASK
                           into:&thisEvent
                           waitFor:0.0
                           threshold:NX_BASETHRESHOLD];
```

Although the highest priority level is 30, the highest threshold that you can specify is 31. This threshold blocks all procedure calls between events.

At the other extreme, assigning a procedure a priority level of 0 effectively blocks it from ever being called between events; the main event loop runs at a threshold of 1.

Note: The priority level of a procedure is compared only to the threshold, not to the priorities of other procedures. If two procedures have priorities at or above the threshold, it doesn't matter that one may have a priority of 29 and the other a priority of 10. Both will be scheduled equally.

Using Timer Events

The modal **mouseDown:** method illustrated under "Coordinating Mouse Events" at the beginning of this section reacted whenever a mouse-up or mouse-dragged event was received. Between events it did nothing, except wait for the next event.

On occasion, however, a modal event loop needs to react even when no event has been received from the Window Server. The absence of an event may indicate that the user is still in the midst of an action, one that generated an earlier event but hasn't produced any new ones. Typically, this occurs while the user is keeping the mouse stationary and holding a mouse button down in order to:

- Press an object (such as a button) that has a repeating or continuous action.
- Automatically scroll the contents of a View after dragging outside it.

Because the mouse isn't moving and the mouse button isn't going up or down, these actions don't generate events. Nevertheless, the modal event loop must continue to respond to the action as if events were being received. It does this by making sure that it will continue to get a stream of events even if none are being generated by the Window Server. The events it arranges for are called *timer events* because they come at regular intervals.

The **NXBeginTimer()** function starts up a timed entry that puts timer events in the event queue at specified intervals. In the example below, it asks for timer events every 0.05 seconds after a delay of 0.1 second:

```
NXTrackingTimer myTimer;
NXBeginTimer(&myTimer, 0.1, 0.05);
```

The first argument to **NXBeginTimer()** is a pointer to an **NXTrackingTimer** structure, defined in the Application Kit's **timer.h** header file. This structure is for the internal use of the timed entry; you don't have to initialize it. If you pass a NULL pointer, memory will be allocated for the structure. Since timer events are usually needed only within a modal event loop, it's generally better to declare the structure as a local variable on the stack as shown above. This avoids the expense of calling **malloc()** to get memory for it.

NXEndTimer() stops the flow of timer events:

```
NXEndTimer(&myTimer);
```

NXBeginTimer() returns a pointer to the **NXTrackingTimer** structure it uses, so even if you pass it a NULL pointer, you'll have access to the pointer required by **NXEndTimer()**.

Timer Example

The following code shows how the earlier **mouseDown:** example would change to include timer events. That example focused on highlighting and unhighlighting an object depending on the location of the cursor. This example focuses on autoscrolling the contents of a View when the user drags outside it.

```
- mouseDown: (NXEvent *)thisEvent
{
    int             shouldLoop = YES;
    int             oldMask;
    NXTrackingTimer myTimer;
    NXEvent        *nextEvent, *lastEvent;

    oldMask = [window addToEventMask:NX_LMOUSEDRAGGEDMASK];
    lastEvent = thisEvent;
    NXBeginTimer(&myTimer, 0.05, 0.05);

    while (shouldLoop) {
        nextEvent = [NXApp getNextEvent:(NX_LMOUSEUPMASK
                                    | NX_LMOUSEDRAGGEDMASK
                                    | NX_TIMERMASK)];
        switch (nextEvent->type) {
            case NX_LMOUSEUP:
                shouldLoop = NO;
                break;
            case NX_LMOUSEDRAGGED:
                lastEvent = *nextEvent;
                break;
        }
    }
}
```

```

        case:NX_TIMER:
            [self autoscroll:&lastEvent];
            break;
        default:
            break;
    }
}

NXEndTimer(&myTimer);
[window setEventMask:oldMask];
return(self);
}

```

It isn't necessary to change the Window's event mask to get timer events, since they're not sent across the connection from the Window Server. You do have to include them in the mask given to **getNextEvent:**, however.

The **autoscroll** method only scrolls when the cursor is outside the receiving View's frame rectangle. It finds the location of the cursor from the event record of the mouse event passed as an argument. Since **autoscroll** messages are sent in response to timer events, the last true mouse event is cached in the **lastEvent** variable.

Avoiding Spin Loops

With timer events, you can write modal event loops that respond only to events; between events, the loop gives up control of the CPU to other applications. This is exactly the behavior required by a multitasking environment.

In a multitasking environment, each application must cooperate with other applications and share processing time with them. You should never write a modal loop that constantly executes instructions without pause, if those instructions do no useful work. For example, a loop that repeatedly peeked for the next event without waiting for one to appear in the queue would spin uselessly between events. As it spins, the loop would arrogate to itself system resources that may be in demand by other tasks. All applications suffer when this happens, including the spinning application.

A modal loop that spins without pause is also at the mercy of the processor it's being run on. On a fast processor, the loop will be executed quickly; on a slower processor it will take more time. A timer event, on the other hand, paces the application's response to a steady user action at the same rate on all processors.

Modal Windows

Sometimes an application needs to set up a modal event loop at the Window level, one step removed from the objects that are actually responding to the events. This can be done through the **runModalFor:** method:

```
[NXApp runModalFor:myWindow];
```

The Window, **myWindow**, should be an attention panel, and it must have an event mask that accepts key-down events (so that it can become the key window).

The **runModalFor:** method puts the panel on-screen, in front of even the main menu, and makes it the key window. It then sets up an event loop that filters the events the Application object receives from the Window Server. It distributes mouse events only if they're associated with the panel; other mouse events are removed from the queue and don't generate event messages. This means that the user can't use the mouse to select any other window in the application (though it can be used to move windows and activate another application).

In this way the panel can command the user's attention until some condition is met—usually the user clicking on one of the panel's buttons.

Breaking the Loop

The modal loop that **runModalFor:** begins can be terminated by sending the Application object a **stopModal** message:

```
[NXApp stopModal];
```

It can also be terminated with an **abortModal** message:

```
[NXApp abortModal];
```

If the response to an event includes a **stopModal** message, the Window's event loop will be broken. The **runModalFor:** method won't attempt to get another event, but it will finish responding to the current one. In contrast, an **abortModal:** message breaks the event loop immediately. Without returning, it raises an exception that causes the **runModalFor:** method to return at once.

Use **abortModal** rather than **stopModal** to break the loop from within code that executes between events—such as a timed entry or a method that responds to a remote message. In these cases, **stopModal** won't work, because **runModalFor:** will check the exit condition only after getting and responding to one more event.

Return Codes

It's often necessary to know why a Window's modal loop has ended. For example, if an attention panel displays three different buttons, any of which can dismiss the panel, the function or method that placed the panel on-screen may want to know which button the user clicked. By using a third method to break the event loop, **stopModal:**, you can pass a return code that identifies the reason why the modal loop is being terminated. Usually the code is an arbitrary integer that identifies the button that was clicked:

```
[NXApp stopModal:2];
```

This integer is passed to and returned by the **runModalFor:** method.

```
int why;
why = [NXApp runModalFor:myWindow];
```

When the modal loop is terminated by an **abortModal** message, **runModalFor:** returns an integer identified by the NX_RUNABORTED constant. When **stopModal** (without a colon) terminates the loop, **runModalFor:** returns NX_RUNSTOPPED.

Only the **stopModal:** method (with a colon) permits you to specify your own return codes. The code should be an integer above -1000; integers less than -999 are reserved by NeXT.

Modal Sessions

On occasion, applications need to carry out time-consuming operations that are not responsive to events—an extensive calculation, perhaps, or moving a large amount of data from one location to another. From the point of view of the user interface, these operations can be divided into three categories:

- Those that prevent the user from doing anything else, but that might be interrupted, aborted, or otherwise controlled by the user.
- Those that prevent the user from doing anything else and cannot be interrupted or aborted.
- Those that shouldn't prevent the user from carrying out other tasks.

Operations in the third category should be placed in a separate Mach thread and performed in the background. Those in the first two categories require you to run an attention panel while carrying out the operation. For the first category, the panel would give the user some control over the operation—notably the ability to terminate it. For the second category, the panel would let the user know what was happening, but would offer no opportunity to abort it.

The **runModalFor:** method can't run this type of panel. Since the modal event loop it sets up controls all the application's activities, it would be impossible to carry out the concurrent operation.

Instead, the application must carry out the operation within a modal session bracketed by **beginModalSession:for:** and **endModalSession:** messages to the Application object. While in the session, a modal event loop—similar to the one set up by **runModalFor:**—is run repeatedly for short periods of time. The rest of the time is available for the concurrent operation. The method that runs the loop, **runModalSession:**, gets events as long as there are any in the event queue, and then returns. An application should send **runModalSession:** messages often enough, at least two or three times a second, to be responsive to the user.

A modal session is identified by an NXModalSession structure. A pointer to the structure is passed as an argument to all three methods mentioned above. The structure isn't one that you need to initialize. If you pass a null pointer to **beginModalSession:for:**, it will create the structure for you and return a pointer that you can pass to **runModalSession:** and **endModalSession:**. However, it's more efficient to declare the structure as a local variable and avoid the necessity of allocating memory for it.

An example of how these methods might be used is given below:

```
NXModalSession theSession;

[NXApp beginModalSession:&theSession for:myWindow];
while ( someCondition ) {
    if ( [NXApp runModalSession:&theSession] == NX_RUNSTOPPED )
        break;
    /* code that performs the concurrent operation */
}
[myWindow orderOut:self];
[NXApp endModalSession:&theSession];
```

As the example shows, **runModalSession:** returns the codes set by **stopModal:**, **stopModal**, and **abortModal**, just as **runModalFor:** does. However, **runModalSession:** usually returns when there are no more events to process, not because the loop has been broken. When it returns without being stopped or aborted, its return value is **NX_RUNCONTINUES**.

Periodic **runModalSession:** messages are required even if the panel isn't one that accepts the users events—for example a panel without buttons that simply informs the user of a lengthy operation that can't be aborted. The messages have three purposes in addition to allowing users to manipulate controls within the panel.

- They clear unwanted events out of the event queue.
- They permit the application to respond to system-defined and kit-defined events, including application-deactivated and window-moved subevents. They thus keep the application responsive to the user.
- They permit the application to receive and respond to remote messages and to execute timed entries.

Drawing in the View Hierarchy

An application draws through its View objects. Each View has:

- An area of the screen, a frame rectangle, where it can draw
- A coordinate system within which it can draw
- A method, **drawSelf::**, that does the drawing

To get a View to draw, you send it a message to display itself. For example:

```
[myView display];
[myView display:&rect :1];
```

Under certain circumstances (described later in this section), the Application Kit generates its own display messages to Views.

The display message brings the View into focus by constructing a PostScript clipping path around its frame rectangle and making its coordinate system the current coordinate system for the application. It then has the View perform its **drawSelf::** method. The display method repeats these steps for each of the View's subviews. The **display** and **display::** messages in the example above would each display **myView** and all the Views below it in the view hierarchy.

Objects that are displayed only through their subviews perform an empty version of **drawSelf::** inherited from the View class. Objects that do any of their own drawing implement a version of **drawSelf::** that overrides the default. **drawSelf::** is always performed indirectly, in response to a display message; it should never appear in a direct message to a View.

The **drawSelf::** method defines a View's static appearance on the screen. Views can also add other methods for *dynamic drawing* in response to the user's actions. These methods might be used to highlight the View, drag it from one place to another, or animate it. They draw outside the display mechanism outlined above. You must first bring the View into focus with a **lockFocus** message, then send it a message to perform the dynamic-drawing method.

The following sections describe focusing, View coordinate systems, **drawSelf::**, **lockFocus**, the display methods, and other aspects of drawing on the NeXT computer.

View Coordinate Systems

As discussed under “View” in the previous chapter, each View’s coordinate system is tied to the location and orientation of its frame rectangle:

- The point that locates the View in its superview’s coordinate system, **frame.origin**, becomes the origin of the View’s own default coordinate system. When the View is brought into focus, this point is made the origin of the current coordinate system.
- The x- and y-axes of the View’s default coordinate system are parallel to the sides of its frame rectangle. If the frame rectangle has been rotated, its default coordinate system rotates with it. When the View is brought into focus, the current coordinate system is rotated around **frame.origin** so that it matches the rotation of the frame rectangle.

Tying the View’s coordinate system to the location and orientation of its frame rectangle has some far-reaching consequences for how Views are displayed. Three of the most important are listed below:

- Each View’s coordinate system is a transformation of its superview’s. This follows from **frame** being defined in the superview’s coordinates.
- None of a View’s drawing instructions (in **drawSelf::** or the dynamic-drawing methods) need to be aware of the View’s location or orientation. Since coordinate values are interpreted relative to the View frame rectangle, a View doesn’t compensate for its movement or rotation on the screen when it draws.
- Changes in a superview’s coordinate system are passed through to its subviews. If a View scales its drawing coordinates, for example, all its subviews will grow or shrink accordingly.

The default coordinate system was illustrated in Figure 6-8, “Default Coordinates,” in the previous chapter. Figure 7-7, below, illustrates it in an even more diagrammatic way. It shows that **frame.origin** becomes (0.0, 0.0) in the View’s default bounds rectangle.

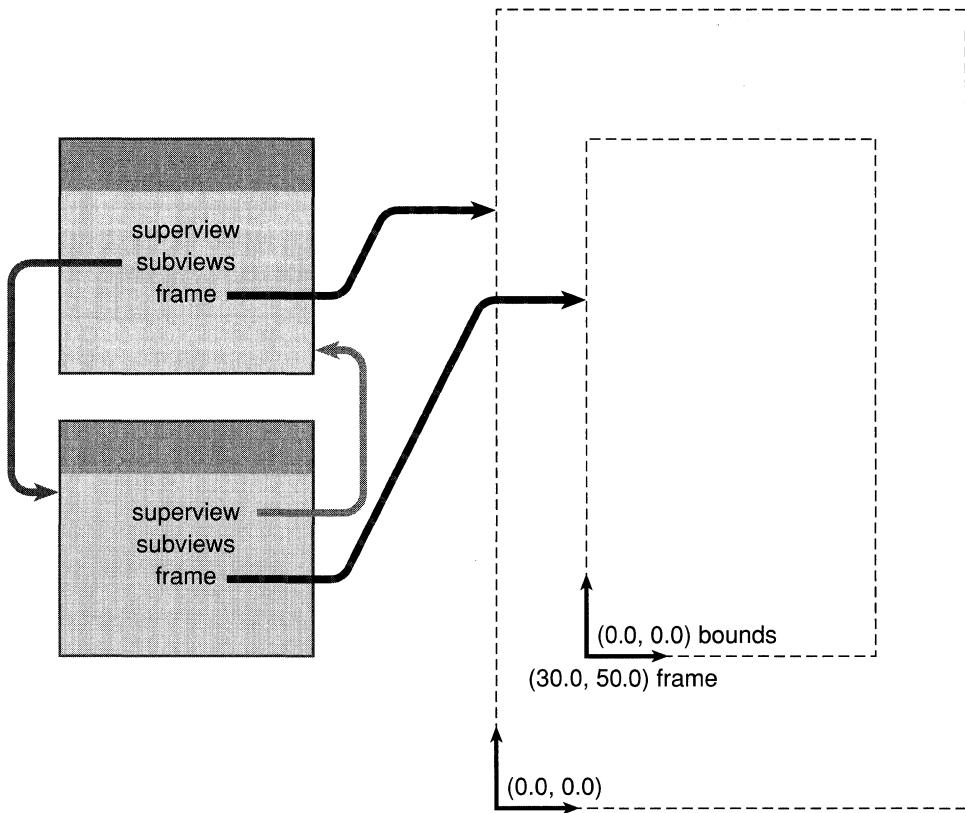


Figure 7-7. Default Coordinates of an Unrotated View

The bounds rectangle expresses the View's location and size in its own drawing coordinates. In Figure 7-7 above, the bounds and frame rectangles enclose exactly the same area. The only difference is that **bounds** records the rectangle in the reference coordinate system that the View uses for drawing inside the rectangle, and **frame** records values that define the rectangle in the superview's coordinate system. In this diagram, and similar ones that follow, **bounds** values are placed inside the subview's frame rectangle, where the subview will draw; **frame** values are placed outside the subview's frame rectangle, in the environment provided by the superview.

The default coordinate system illustrated in Figures 6-8 and 7-7 can be altered through methods that translate, scale, and rotate it, or that flip the polarity of its y-axis.

Flipping the Coordinate System

By default, Windows and Views inherit the orientation of the screen coordinate system; the positive x-axis extends rightward and the positive y-axis extends upward. Views (but not Windows) can opt to flip this coordinate system so that the positive y-axis extends downward.

```
[myView setFlip:YES];
```

The **frame** instance variable of a flipped View is no different from that of a View that isn't flipped: The width and height of the View are expressed by positive values, and the point locating the View in its superview's coordinates is the one with the smallest x and y values.

The **bounds** instance variable of a flipped View also remains the same: Its width and height are expressed by positive values, and it records the rectangle corner with the smallest x and y values in the View's reference coordinate system. Because the polarity of the y-axis is flipped, this corner will (barring rotation) be the upper left corner of the View, rather than the lower left corner.

Figure 7-8 illustrates the same two Views as Figure 7-7, except that here the subview's coordinate system has been flipped.

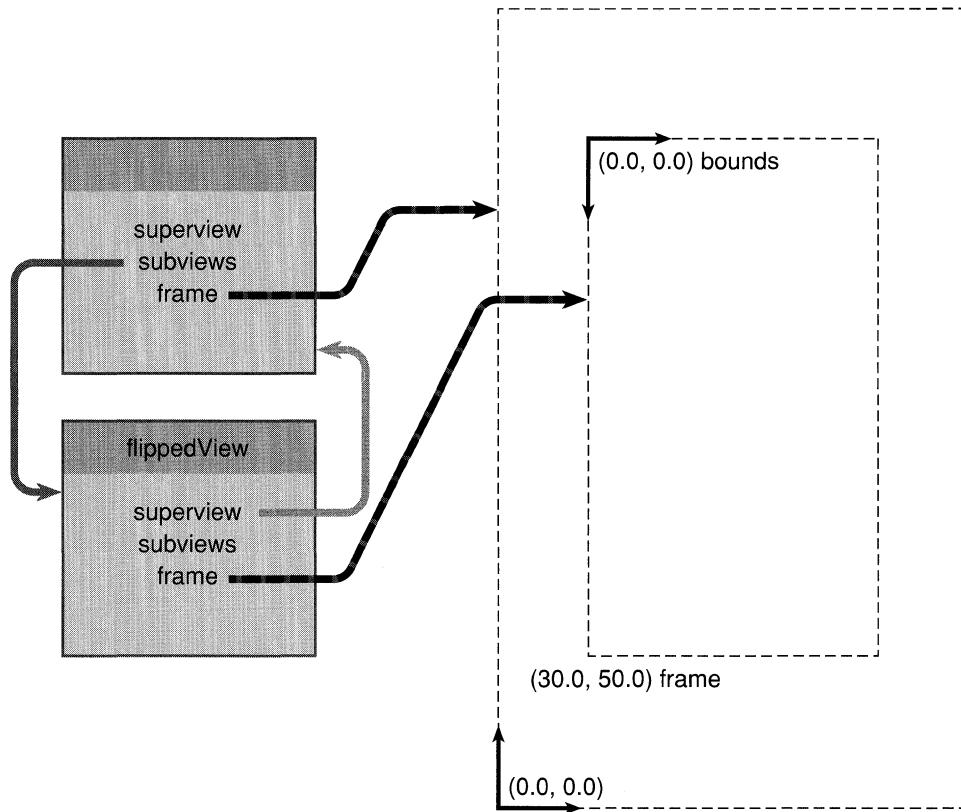


Figure 7-8. Flipped Coordinates

Note that the frame rectangle continues to locate the View in its superview's unflipped coordinate system, but the bounds rectangle in which the View draws has been flipped. **frame.origin** and **bounds.origin** are no longer located at the same point.

Flipped coordinates are an inherent property of the View, not a transient feature that can be turned on and off. A View should receive only one **setFlip:** message during its life, before it's first displayed, preferably in the class method that creates it. All of the View's drawing should assume its flipped coordinates.

The **isFlipped** method returns whether or not the receiving View is flipped:

```
BOOL flipState;
flipState = [myView isFlipped];
```

Transitivity

Flipped coordinates are an exception to the rule stated above, that transformations of a superview's coordinate system carry over to its subviews. Flipping a parent View doesn't flip its subviews. A View won't become flipped when it's made the subview of a flipped View, and a flipped View won't flip again, back to the original orientation, if it's made the subview of another View with flipped coordinates. A View has flipped coordinates *only* if it receives a **setFlip:** message with YES as the argument.

Unflipping

The argument passed to **setFlip:** is almost always YES. Passing NO as the argument would unflip the View, but Views are unflipped by default, and once a View has been flipped, it should stay that way.

There's just one situation where NO is an appropriate argument to **setFlip:**. If you define a subclass and inherit a class method that includes a **setFlip:** message to flip the instances it creates, you'll need to cancel that message with another **setFlip:** message to make objects belonging to the subclass unflipped.

For example, if this is the superclass method,

```
+ newView
{
    self = [super new];
    [self setFlip:YES];      /* flips its coordinates */
    . .
}
```

the subclass method might look like this:

```
+ newView
{
    self = [super newView];  /* inherits the flip */
    [self setFlip:NO];      /* cancels it */
    . .
}
```

Drawing Text

Flipped coordinates are mainly useful to objects that draw multiple lines of text. Coordinate values increase, rather than decrease, as more lines are added from the top of the View to the bottom. In the Application Kit, the Text and TextField objects both use flipped coordinates, as do Matrices, Forms, Buttons, Sliders, Scrollers, and ScrollViews. (But a View doesn't have to be flipped to be scrollable.)

When drawing text in a View with flipped coordinates, you must use a font with a matrix that flips the y-axis of the characters that are drawn. Fonts have higher y-coordinate values at the top of the character outline and lower y-coordinate values at the bottom (with values of exactly 0.0 at the baseline). Unless the font matrix flips this polarity to match the flipped View, the characters will be displayed upside down.

A flipped matrix is specified when first setting the font through the **newFont:size:style:matrix:** method:

```
id myFont;
myFont = [Font newFont:"Courier-Bold"
          size:12.0
          style:0
          matrix:NX_FLIPPEDMATRIX];
```

The constant NX_FLIPPEDMATRIX is defined in the interface file for the Font class. It specifies a font matrix with these six values:

```
1.0, 0.0, 0.0, -1.0, 0.0, 0.0
```

An unflipped matrix is specified by another constant, NX_IDENTITYMATRIX. An identity matrix has only positive values:

```
1.0, 0.0, 0.0, 1.0, 0.0, 0.0
```

If the text you draw appears upside down, the culprit is likely to be the wrong matrix.

Notifying Superviews

Sometimes a View needs to let its ancestors know that it's flipped so they can adjust the display accordingly. (In the Application Kit, this is important mainly to the ClipView that contains a scrollable document View.)

If a View receives a **notifyWhenFlipped:** message with YES as the argument,

```
[myView notifyWhenFlipped:YES];
```

it will send its superview a **descendantFlipped:** message when **setFlip:** flips its coordinates. The default implementation of **descendantFlipped:** simply passes the message on to the receiver's superview, so the message will simply climb the view

hierarchy until it runs out of Views. A View that needs to know whenever one of its descendants is flipped can override the default to do whatever is necessary. The argument passed in **descendantFlipped:** messages identifies the flipped View.

Modifying Default Coordinates

A View can alter its coordinate system with methods that parallel the standard PostScript transformations:

```
NXCoord x, y, width, height;  
float angle;  
[myView translate:x :y];  
[myView scale:width :height];  
[myView rotate:angle];
```

- **translate::** moves the View's coordinate origin to (x, y).
- **scale::** makes the View's x-coordinate unit equal to **width**, and its y-coordinate unit equal to **height**.
- **rotate:** turns the View's coordinate axes by **angle** from their present angle of orientation.

Like the PostScript operators, these methods affect the coordinate system incrementally. If a View is sent a message to rotate 60°, sending it another message to rotate 60° turns it 120° from its original orientation.

Unlike the PostScript operators, however, they don't have an immediate effect on the current coordinate system. Alterations to a View's coordinate system take effect only when it's next brought into focus.

Three other methods modify a View's coordinate system to a more absolute specification:

```
[myView setDrawOrigin:x :y];  
[myView setDrawSize:width :height];  
[myView setDrawRotation:angle];
```

- **setDrawOrigin::** translates the View's coordinate system so that (x, y) designates the same point on-screen as **frame.origin**.
- **setDrawSize::** scales the View's coordinate system so that **width** and **height** describe the size of the frame rectangle.
- **setDrawRotation:** rotates the View's coordinate system so that **angle** is the difference between its frame and the coordinate axes it uses for drawing.

These three methods are not incremental. If a View is sent a `setDrawRotation:` message to rotate 60°, sending it the same message again would leave its coordinate system at the same angle of orientation, just 60° from the angle of orientation of its frame.

The six methods listed above affect only the coordinate system used for drawing; they don't affect the location, size, or rotation of the View's frame rectangle.

Rotated Bounds

Although scaling and translating a View's coordinate system affect the values stored in its `bounds` instance variable, the area covered by the bounds rectangle remains identical to the area covered by the frame rectangle. Rotating a View's coordinates, on the other hand, can change the orientation of the bounds rectangle so that it no longer matches the frame rectangle. Unless the rotation is an exact multiple of 90°, the bounds rectangle will be somewhat larger than the frame rectangle. This is illustrated in Figure 7-9.

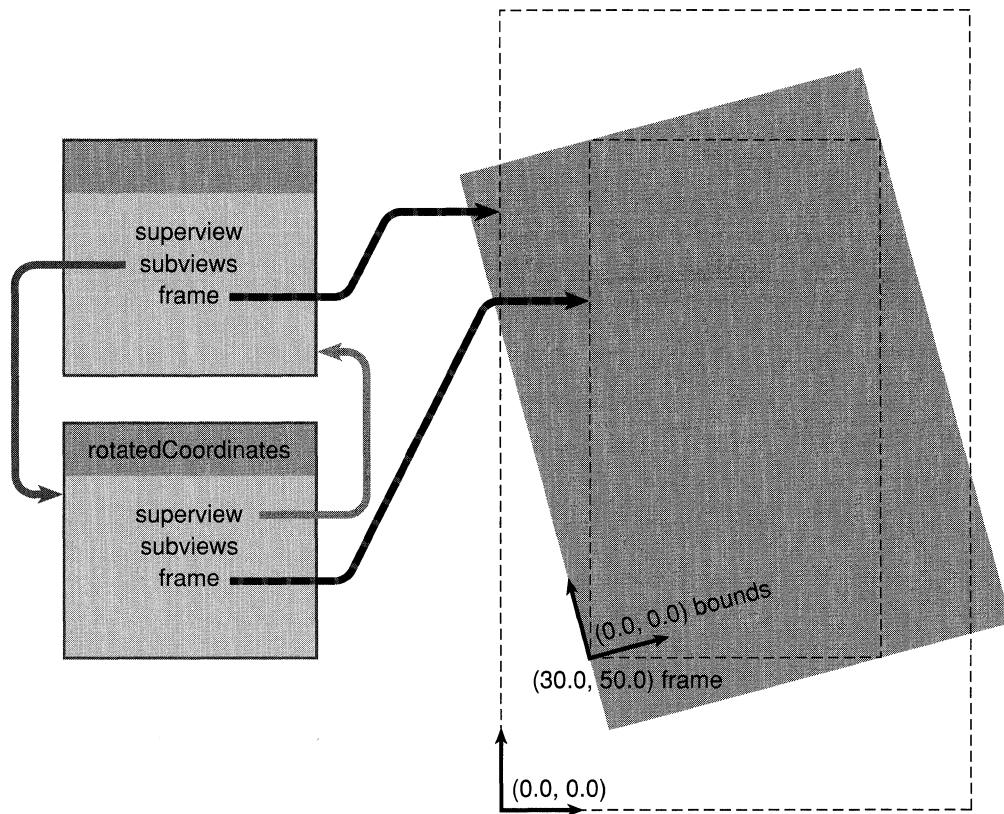


Figure 7-9. Rotated Bounds Rectangle

Figure 7-9 shows a View whose drawing coordinates have been rotated counterclockwise 15°. The bounds rectangle is defined in the rotated coordinates; the sides of the rectangle are parallel to the rotated x- and y-axes. This rectangle must be larger than the frame rectangle for it to encompass all of the View.

The **boundsAngle** method returns the rotation of the bounds rectangle relative to the frame rectangle. For the example in Figure 7-9, it would return 15.0 degrees.

```
float rotation;  
rotation = [myView boundsAngle];
```

The **setDrawOrigin::** and **setDrawSize::** methods specify values that describe a View's frame rectangle in the View's own coordinate system. They therefore usually set the values in the **bounds** instance variable, since the bounds rectangle usually designates the same area as the frame rectangle. However, if the View has a rotated coordinate system, the bounds rectangle won't match the frame rectangle, so **bounds** won't have the values set by these methods.

Since the visible rectangle and the bounds rectangle are both stated in the View's reference coordinate system, the visible rectangle rotates with the bounds rectangle. This is illustrated in Figure 7-10.

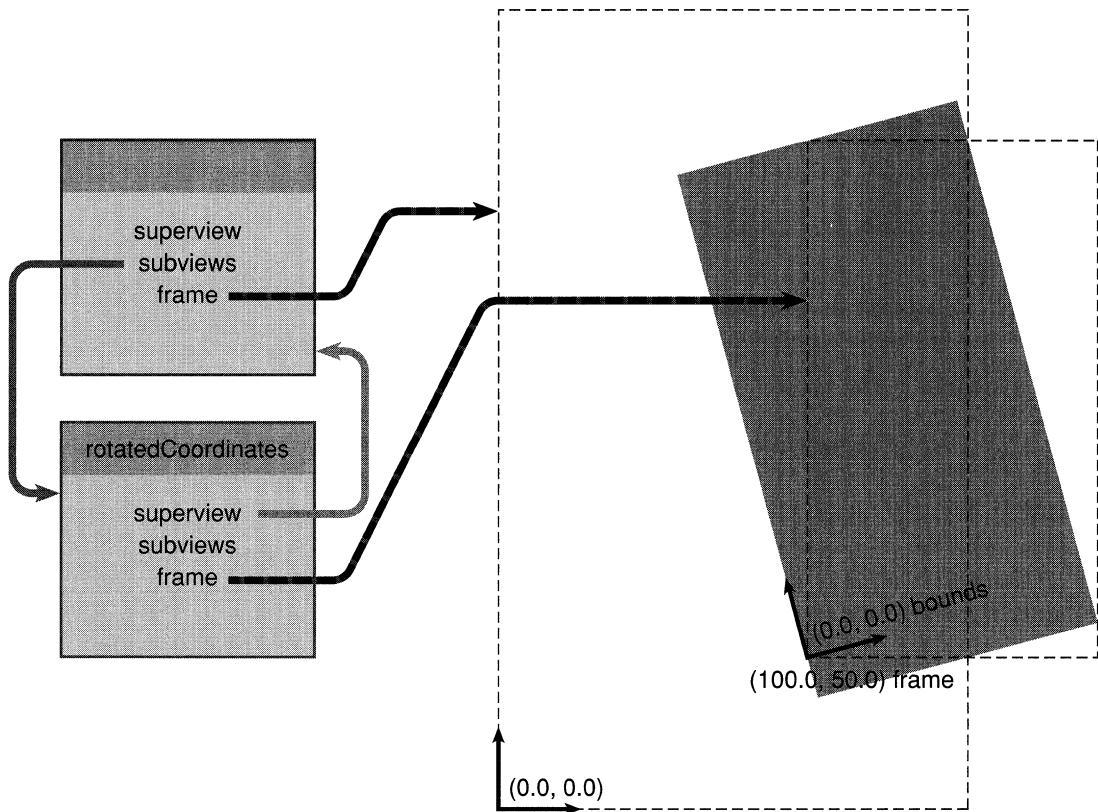


Figure 7-10. Rotated Visible Rectangle

For a rotated rectangle to enclose all the visible area of a View, it must also contain some areas that are not visible and some that may not even be in the View. Considerations of efficiency would therefore recommend against rotating the reference coordinate system for a View, but to do any rotation necessary within the View's drawing code.

Transitivity

Modifications to a View's coordinate system can affect the size and placement of its subviews. They can also affect the coordinate system that a subview draws in.

By default, subview coordinate systems reflect the superview's rotation and scaling. But because each subview's default coordinate system is translated to the point designated by `frame.origin`, the subview overrides any translation done by the superview.

Two methods let you know whether a View's coordinate system has been rotated or scaled, either directly or through any ancestor Views above it in the view hierarchy:

```
BOOL rotated, modified;  
  
rotated = [myView isRotatedFromBase];  
modified = [myView isRotatedOrScaledFromBase];
```

If these methods return NO, the View has the orientation and scaling of the base and screen coordinate systems.

Basic and Temporary Coordinate Systems

setFlip: and the six methods that transform a View's coordinate system should be used only to set up the basic coordinate system for the View, the one that's the starting point for the drawing instructions in the View's `drawSelf::` method.

It's typical, while drawing in the PostScript language, to alter the coordinate system repeatedly for temporary effects—to scale it to draw an oval, to rotate it to draw text at an angle, or to repeatedly translate it so that a single procedure can draw a figure in more than one position. These temporary changes to the coordinate system should be done directly in PostScript code, through `pswrap`-generated functions called by `drawSelf::` (or by the dynamic drawing methods). They shouldn't be done through View methods. See "Drawing Methods," below, for more on how drawing methods like `drawSelf::` can change the graphics state.

Converting Coordinates

Because each View has its own coordinate system, it's often necessary to convert a point or an area specified in one View's coordinate system to the coordinate system of another View. The View class defines methods that can convert NXPoint, NXSize, and NXRect structures from one View to another:

```
[myView convertPoint:&point toView:anotherView];
[myView convertSize:&size toView:anotherView];
[myView convertRect:&rect toView:anotherView];
```

These methods assume that the first argument refers to a structure with values in the receiving View's (**myView**'s) coordinate system. They alter those values to the coordinate system of the second argument (**anotherView**), provided the two Views belong to the same window. If the second argument is **nil**, the conversion is to the window's base coordinate system. On-screen locations and areas aren't altered, only the coordinate systems that they're expressed in.

Three other methods convert in the opposite direction, from the coordinate system of a specified View to the coordinate system of the receiver:

```
[yourView convertPoint:&point fromView:secondView];
[yourView convertSize:&size fromView:secondView];
[yourView convertRect:&rect fromView:secondView];
```

If the View specified in the second argument is **nil**, the three methods above convert the NXPoint, NXSize, or NXRect structure from the base coordinate system to the receiving View's coordinate system.

There are also methods that optimize for the special case when one View is the other View's superview:

```
[myView convertRectFromSuperview:&point];
[myView convertRectToSuperview:&rect];

[yourView convertPointFromSuperview:&point];
[yourView convertPointToSuperview:&rect];
```

The Window class defines methods for converting an NXPoint structure from the base coordinate system to the screen coordinate system, and from the screen coordinate system to the base coordinate system:

```
[myWindow convertBaseToScreen:&point];
[myWindow convertScreenToBase:&point];
```

Focusing on a View

Before a View draws, it's necessary to lock the focus on it:

```
[myView lockFocus];
```

After it's finished drawing, the focus should be unlocked:

```
[myView unlockFocus];
```

Locking the focus ensures that the View draws in the correct window, place, and coordinate system. It makes the View's reference coordinate system the current coordinate system for the application. Unlocking the focus returns to the coordinate system of the View that was previously in focus.

The **isFocusView** method returns whether the focus is currently locked on the receiving View:

```
BOOL canDraw;
canDraw = [myView isFocusView];
```

The Application object's **focusView** method returns the last View that was brought into focus:

```
id currentView;
currentView = [NXApp focusView];
```

If a View draws through the display mechanism, you don't have to explicitly lock and unlock the focus; the display methods perform **lockFocus** and **unlockFocus** for you. For dynamic drawing that's done outside of the display mechanism, however, explicit **lockFocus** and **unlockFocus** messages are required. The display methods are discussed in a later section of this chapter.

How Focusing Works

The Application Kit takes the following steps to bring a View into focus:

1. It constructs a clipping path around the View. This ensures that the View won't draw outside its frame rectangle.
2. It makes the View's coordinate system the current coordinate system for the application, as recorded in the current transformation matrix (CTM) of the current graphics state.

Views are brought into focus by working down the view hierarchy, from superview to subview. Since each View keeps track of its own coordinate system as a modification of its superview's, a View can be brought into focus only after its superview. To focus on the superview, the superview's superview might first have to be brought into focus, and so on

all the way up to the View at the root of the view hierarchy (the frame view). If a View must be focused from scratch, the Application Kit begins with the base coordinate system of the window and frame view, brings the content view into focus, then brings a subview of the content view into focus, and continues to work down the hierarchy to the target View.

The Application Kit doesn't take these steps if they're unnecessary; the focusing mechanism has been optimized and is quite efficient. Focusing begins with the window's base coordinate system only if there's no View with a determinate coordinate system closer to the target View. Usually there is, either the View that's currently in focus or a View with a coordinate system specified by its own graphics state object. (Assigning graphics state objects to Views is discussed below under "Drawing Methods.")

Because each View is clipped to its frame rectangle, focusing down the view hierarchy means that a subview can't draw outside the area allotted to its superview. In the PostScript language, each additional clipping path only further constrains the area where images can appear, so if a subview lies outside the frame rectangle of *any* ancestor View, it won't be displayed.

Clipping

Of the steps that are taken to bring a View into focus, clipping to its frame rectangle is perhaps the most time-consuming. A View that doesn't require a clipping path to ensure that it won't draw outside its allotted area can skip this step by sending itself a **setClipping:** message with NO as the argument:

```
[self setClipping:NO];
```

As the Application Kit focuses from superview to subview down the view hierarchy, whenever it encounters a View that has received this message, it skips the step that constructs a clipping path around the View's frame rectangle.

A View should avoid clipping only if it can be assured that it and all its subviews won't attempt to draw outside its frame rectangle. Many of the View subclasses defined in the Application Kit—Button, Text, Scroller, and ScrollView—don't clip.

The **doesClip** method returns whether the receiver will be clipped during focusing:

```
BOOL clips;
clips = [myView doesClip];
```

Like **setFlip:**, **setClipping:** determines a permanent property of a View. A View should receive the **setClipping:** message before it draws or modifies its coordinate system; it's best to include it in the class method that creates the View. Once clipping has been turned off, it shouldn't be turned back on again.

However, if you define a subclass and inherit a class method with a **setClipping:** message that turns clipping off, you can cancel that message by sending another **setClipping:** message in the subclass's method, this time with YES as its argument.

```
+ newView
{
    self = [super new]; /* the new method that prevents clipping */
    [self setClipping:YES];/* the remedy */

    .
}

}
```

Using theSuperview's Coordinates

A View can opt to use the same coordinate system as its superview:

```
[myView drawInSuperview];
```

This avoids the overhead of transforming the superview's coordinate system to focus on the View. It also makes the View's **bounds** instance variable identical to **frame**, as shown in Figure 7-11.

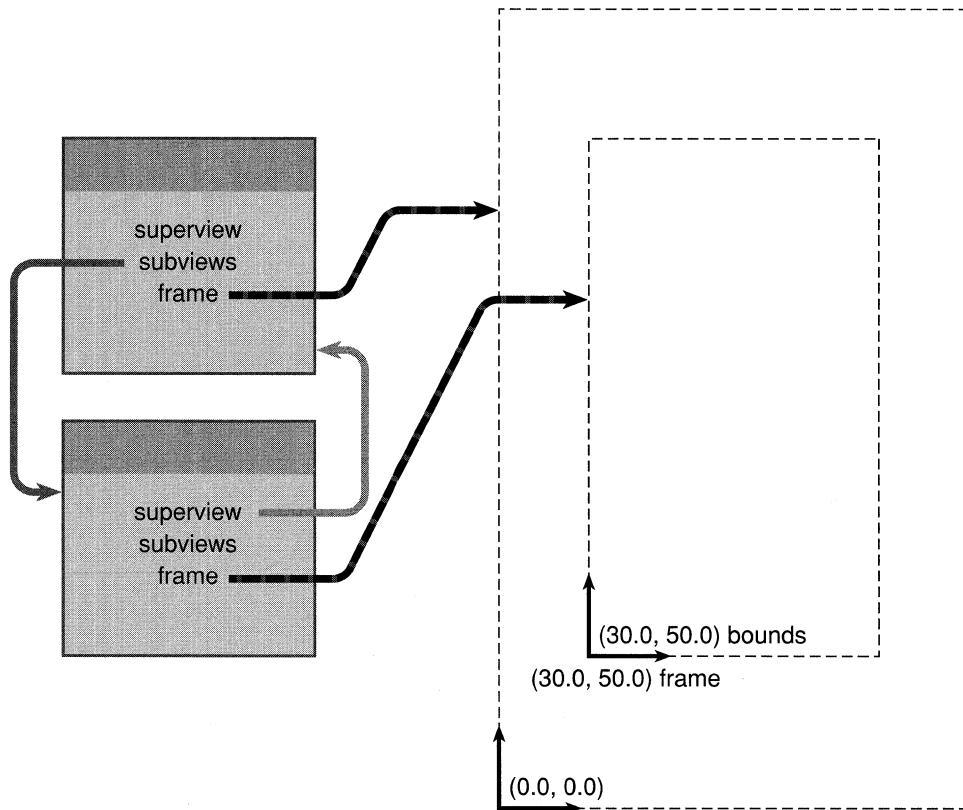


Figure 7-11. Drawing in theSuperview's Coordinates

Because its superview's coordinate system doesn't have to be transformed to bring the View into focus, less PostScript code is sent to the Window Server. A View that receives both **drawInSuperview** and **setClipping:** messages can avoid both steps that are taken to focus on it from its superview, the clipping path and coordinate transformations.

After receiving a **drawInSuperview** message, a View's drawing instructions must take into account the View's location within its superview (**bounds.origin**) and the width and height of its frame rectangle (**bounds.size**).

Like **setFlip:** and **setClipping:**, **drawInSuperview** establishes an inherent attribute of the View, one that should be set as soon as the View is created and shouldn't be changed thereafter. In particular, a **drawInSuperview** message shouldn't be sent to a View that also receives messages to transform its default coordinate system.

The Text object is the only object defined in the Application Kit that draws in its superview's coordinate system.

Modifying drawInSuperview Coordinates

If a View receives a **drawInSuperview** message after its coordinate system has been scaled or rotated, the message will have no effect. If it receives the message after it has been translated (but not scaled or rotated), the message will have no effect until the translation is reversed.

The sole purpose of a **drawInSuperview** message is to avoid the overhead of having to focus on the View independently of its superview. Once it has been translated, scaled, or rotated, a View has its own coordinate system; there's no point in sending it a **drawInSuperview** message.

Nevertheless, if messages to modify the View's coordinate system are sent after the **drawInSuperview** message, they modify the coordinate system that **drawInSuperview** established for the View. (Despite the fact that the View was using its superview's coordinate system, the modifications affect only the View itself, not its superview.)

Flipping drawInSuperview Coordinates

A View can both draw in its superview's coordinates and be flipped:

```
[myView drawInSuperview];
[myView setFlip:YES];
```

The Text object is such a View.

The **setFlip:** message affects only the View that receives it, not its superview. Flipping the coordinates in this way doesn't negate all the effects of **drawInSuperview**; reversing the polarity of the y-axis is the only transformation that's needed to focus on the View independently of its superview.

Similarly, if a View draws in its flipped superview's coordinates, it won't itself automatically be flipped. Although the **drawInSuperview** message makes the **bounds** and **frame** instance variables identical, **bounds.origin** and **frame.origin** will refer to two different points. This is illustrated in Figure 7-12.

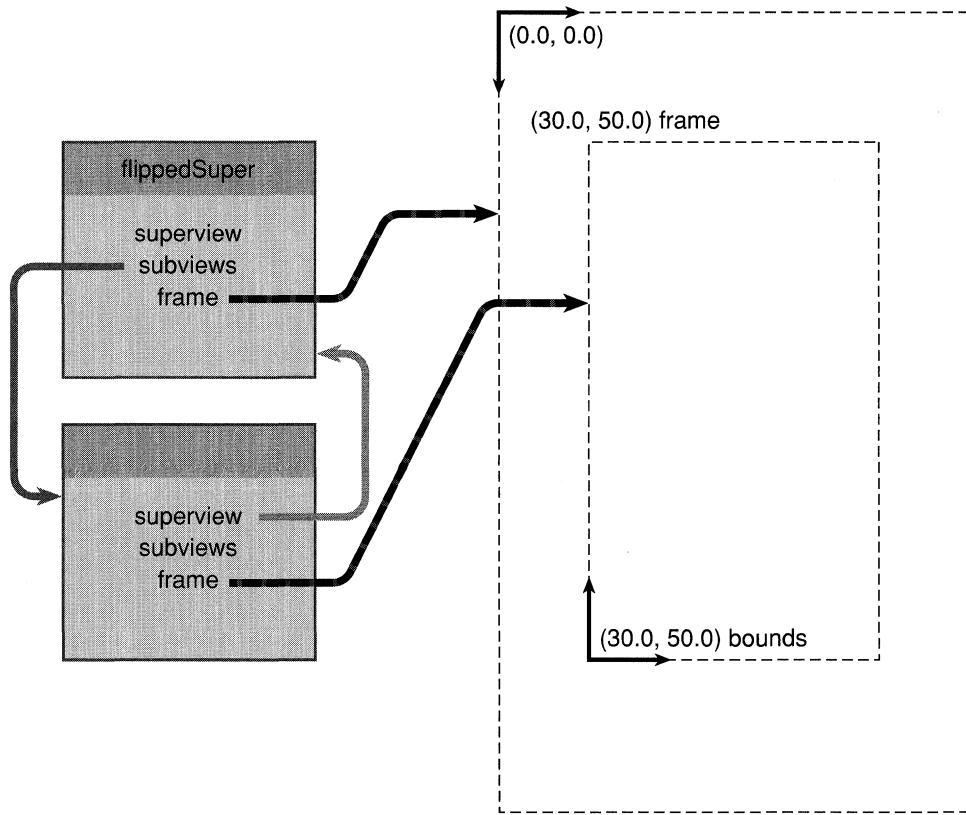


Figure 7-12. Drawing in a Flipped Superview

A View is flipped only if it receives a **setFlip:** message; it can't inherit this feature from its superview, even if it draws in its superview's coordinates. Therefore, to have a subview draw in exactly the same coordinate system as its flipped superview, it must receive both a **drawInSuperview** message and a **setFlip:** message, as illustrated above. This avoids any and all coordinate transformations to focus on the View.

Locking and Unlocking the Focus

A **lockFocus** message makes the receiving View's coordinate system the application's current coordinate system. Each **lockFocus** message must be paired with an eventual **unlockFocus** message to the same View, after the View has finished drawing:

```
[myView lockFocus];
/* drawing code goes here */
[myView unlockFocus];
```

If a View is already in focus when a **lockFocus** message is sent, **lockFocus** saves the current graphics state (with the PostScript **gsave** operator) before bringing the receiving View into focus. The **unlockFocus** message later restores the saved graphics state (with the PostScript **grestore** operator). This returns the focus to the View that had it before the **lockFocus** message.

An **unlockFocus** message can balance only one previous **lockFocus**.

Matching **lockFocus** and **unlockFocus** messages must bracket all the drawing that a View does. Like braces in C code, they can be nested. In the example below, the **doDynamicDrawing** method locks the focus on the receiver (**self**), then locks and unlocks the focus on a companion View before unlocking the focus on the receiver.

```
- doDynamicDrawing
{
    [self lockFocus];
    [self drawFirstPart];
    if ( companionView ) {
        [companionView lockFocus];
        [companionView doOtherDrawing];
        [companionView unlockFocus];
    }
    [self drawSecondPart];
    [self unlockFocus];
}
```

When the companion View receives an **unlockFocus** message, the focus immediately returns (through **grestore**) to **self**.

It's possible to focus repeatedly on the same View, without intervening **unlockFocus** messages. Successive **lockFocus** messages to the same View don't generate any additional PostScript code (except for **gsave**); since the View is already in focus, it doesn't have to be brought into focus again. However, if something in the graphics state has changed or the View has altered its frame rectangle, it will be refocused.

The **lockFocus** method returns boolean YES if it doesn't need to do anything to bring the receiving View into focus. This lets the drawing method know that it has the same graphics state as before, so that it can avoid reinitializing graphics state parameters.

Drawing Methods

The method that draws a View is **drawSelf::**. **drawSelf::** messages are generated when the View, or one its ancestors in the view hierarchy, receives a display message. You must implement a **drawSelf::** method for each custom View that you want to draw according to your own specifications, but you should rely on display messages to perform the method.

Since display messages are often generated by the Application Kit in response to user actions, **drawSelf::** must be able to reach all the code necessary to render the View. Typically, **drawSelf::** calls **pswrap**-generated functions to send PostScript code to the Window Server. It may also send messages to Bitmap objects to composite source images stored in off-screen windows. If a View uses a Cell to do any of its drawing, **drawSelf::** will send messages to perform the Cell's drawing methods, **drawSelf:inView:** and **drawInside:inView:**.

The **drawSelf::** method draws the neutral, static appearance of a View. It can be matched by dynamic drawing methods that temporarily alter the View's appearance in response to events. Messages to perform dynamic drawing methods should be generated within the event-handling code you write.

drawSelf:: and the dynamic drawing methods operate in an environment specific to a View. The View determines the window, the area within the window, and the initial coordinate system for drawing. Focusing on the View makes the current graphics state reflect the View's attributes:

- The current window is the window where the View is located.
- The current clipping path enforces the restricted area within the window where the View can draw.
- The current transformation matrix (CTM) records the View's initial coordinate system for drawing.

These View-specific factors aren't the whole story, however. The current graphics state includes other parameters, such as line width and halftone screen, and there may be further restrictions on where it's appropriate for the View to draw. The sections below discuss the factors that define the drawing environment.

Drawing Rectangles

The two arguments passed to **drawSelf::** indicate how much of the View needs to be drawn. The first argument is a pointer to an array of rectangles (**NXRect ***) specified in the View's reference coordinate system, and the second argument (an **int**) indicates how many rectangles are in the array.

Usually there's just one rectangle in the array, but there may be three. When there are three, the first rectangle is the union of the second and third—that is, it's the smallest rectangle

that completely encloses the other two. Specifying the area that needs to be displayed as the sum of two rectangles is a natural consequence of some user actions, such as resizing a window so that it's larger or scrolling a View at an angle. In Figure 7-13, the two rectangles that need to be displayed after the user has scrolled are shown in white. Their union, the first rectangle in the array, would be a rectangle as large as the View.

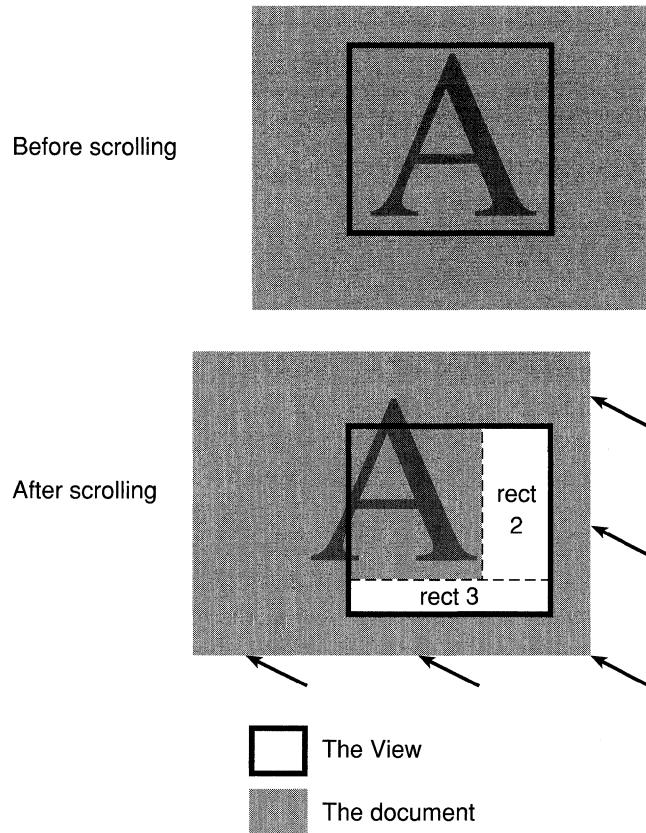


Figure 7-13. Update Rectangles

The drawing rectangles specify an area that, at its largest, is identical to the View's visible rectangle. They often specify an area smaller than the visible rectangle, but in no case do they specify any area falling outside the visible rectangle. Just as the bounds rectangle defines the largest area on the screen that a method should attempt to draw in, the drawing rectangles define the minimal area that should be redrawn. A `drawSelf::` method should be sure to cover at least the area specified in the drawing rectangles it's passed.

A View that's not scrolled and that almost certainly lies within the frame rectangles of all its ancestors can safely ignore the drawing rectangles and draw everything within its bounds rectangle. However, considerations of efficiency require most Views to limit their drawing to the smallest possible area. It's wasteful to redraw areas that don't require updating or to send drawing code to the Window Server if it won't be rendered. Views that are scrolled, especially large Views, should stay as close to the drawing rectangles as possible.

Dynamic drawing methods aren't passed an array of drawing rectangles by the Application Kit. These methods should use information from the event record and the visible rectangle to determine where to draw. The **getVisibleRect:** method supplies the visible rectangle:

```
BOOL      isVisible;
NXRect   drawingArea;

isVisible = [myView getVisibleRect:&drawingArea];
```

getVisibleRect: determines how much of the receiving View's bounds rectangle lies within the frame rectangles of all its ancestor Views. If none of the receiving View lies within its ancestors, none of it is visible and **getVisibleRect:** returns NO. Otherwise, **getVisibleRect:** returns YES and places the View's visible rectangle in the structure referred to by its argument.

Graphics State Parameters

All of a View's drawing methods, including **drawSelf::** and the methods used to do dynamic drawing, assume the View's coordinate system as a starting point. They also assume other aspects of the graphics state—such as the halftone screen, line cap, and clipping path. A drawing method is free to change any graphics state parameter as long as it restores the original value when it's finished. Failure to do so may mean that other drawing methods won't function properly. This can adversely affect the methods defined in Application Kit classes such as Text, Button, and Scroller, as well as the methods you define in View subclasses.

However, not all graphics state parameters have a presumed value. Some—such as the current path, line width, and color—are altered so frequently by Views that there's no point in defining a default value. A drawing method can't rely on an initial value for these parameters; it must be careful to set them to their required values before drawing. It also has no responsibility for restoring their initial values later.

The chart below lists the parameters of the graphics state and their assumed values in the Application Kit.

Parameter	Presumed Value
current transformation	The reference coordinate system for the View matrix (CTM)
color	No presumed value
position	No presumed value
path	No presumed value
clipping path	A path that's constructed around the frame rectangle and the drawing rectangles, as instructed by setClipping: and display::: messages sent to the View and its superviews
font	No presumed value
line width	No presumed value
line cap	The initial PostScript value, 0, for a square butt end
line join	The initial PostScript value, 0, for mitered joins
halftone screen	A device-dependent, type 3 halftone dictionary
flatness	The initial PostScript value, 1.0
miter limit	The initial PostScript value, 10
dash pattern	The initial PostScript value, a normal solid line
device	The current window
stroke adjustment	True
alpha	1.0 (opaque)
instance drawing mode	False

Note: The last two items on this list are NeXT additions to the graphics state. Stroke adjustment is an addition of the Display PostScript system.

There are two recommended ways to restore a value that's been changed. The first is to save the current graphics state before drawing and restore it when you're through:

```
gsave
PostScript code that changes graphics state parameters
grestore
```

The second is to put the current value of the parameter that will change on the operand stack before you start and restore it when you’re done:

currentlinecap% Puts it on the operand stack
PostScript code that changes the line cap
PostScript code that adjusts the stack
setlinecap% Uses the value on the stack

To save and restore the halftone screen, use the **currenthalftone** and **sethalftone** operators (or **gsave** and **grestore**) instead of **currentscreen** and **setscreen**.

The Display Methods

Five methods can be used to display a View and its subviews:

```
[myView display];
[myView display:&bounds :1];
[myView display:&bounds :1 :NO];

[myView displayIfNeeded];
[myView displayFromOpaqueAncestor:&bounds :1 :NO];
```

Collectively, these five methods are referred to as the “display methods.” The first three are standard, general purpose methods. They’re at the heart of the View display mechanism. The second two methods are more specialized. They’re used in common, but quite specific situations.

The first three messages shown in the list above are all equivalent. Although the first two methods are the ones most commonly used, they’re each simplified versions of the third method; **display** and **display::** both work by sending a **display:::** message to **self**.

- The **display::** method passes its two arguments on to **display:::** and adds NO as the third argument.
- The **display** method is the same as **display:::** with a pointer to the receiving View’s visible rectangle as the first argument, 1 as the second argument, and NO as the third argument.

The precise meaning of each argument is discussed in the section titled “Display Method Arguments” below. Most programs can ignore the arguments and safely use **display** in all situations.

The fourth method, **displayIfNeeded**, is like **display**, but it displays only those Views and subviews that have changed since they were last drawn on-screen and therefore need to be redisplayed. It's discussed in the next section, "Managing a Window's Display."

The fifth method, **displayFromOpaqueAncestor:::**, is similar to **display:::**, but it ensures that any Views that draw in the background of the receiving View are also displayed. It's discussed in the section titled "Displaying Background Views" below.

No matter which display method is used, the essential work is done in four steps:

1. The focus is locked on the receiving View.
2. A message is sent for it to perform its **drawSelf::** method.
3. Its subviews are displayed.
4. The focus is unlocked.

The third step makes the display process recursive. If the receiving View has any subviews, the focus is locked on each subview in turn and a message is sent for the subview to perform its **drawSelf::** method. If the subview has subviews of its own, the focus is locked on each of them and they perform their own **drawSelf::** methods. Subviews are picked in the order that they appear in the View's **subviews** list, which usually reflects the order in which they were made subviews of the View. A View is sent an **unlockFocus** message only after all of its descendants have been displayed.

Order of Display

These steps are significant; they mean that Views draw in a particular order:

- Subviews draw on top of (after) their superviews.
- A subview that's further down in the subviews list draws on top of (after) sibling subviews that are earlier in the list.

Note that the order in which **hitTest:** attempts to associate mouse-down events with subviews is exactly opposite to the order of display. Displaying starts at the beginning of the subviews list; **hitTest:** starts at the end. If two sibling subviews overlap on the screen and a mouse-down event occurs in the overlapped area, the subview that draws on top will get the event.

Views can be reordered in the subviews list by methods defined in the View class and described under "The Core Framework" in the previous chapter.

Display Method Arguments

The two arguments to **display:::** (the first two arguments to **display:::** and **displayFromOpaqueAncestor:::**) limit the amount of drawing code that's sent to the Window Server. The first argument is a pointer to an array of rectangles (**NXRect ***) that specify the areas to be displayed; the second argument (an **int**) indicates how many rectangles are in the array. With some modifications, the display methods pass these arguments on to the View's **drawSelf:::** method.

It's assumed that the rectangles are specified in the same coordinate system as the View that receives the display message. There can be 0, 1, or 3 rectangles in the array. If it's 3, the first rectangle should be the union of the second and third, as described under "Drawing Rectangles" above.

If the rectangle pointer (the first argument) is NULL or the number of rectangles (the second argument) is 0, the display methods substitute a pointer to the visible rectangle for the first argument and 1 for the second. The visible rectangle encloses the smallest area that needs to be displayed to guarantee that everything visible in the View is drawn.

If any rectangles are passed to a display method, the method intersects each of them with the View's visible rectangle to ensure, to the extent possible, that they designate only areas lying within the View. The display methods then use the array of rectangles in three ways:

- They pass the rectangles on as arguments to the View's **drawSelf:::** method. **drawSelf:::** can ignore the arguments or use them to optimize the drawing it does.
- They don't display any subviews that lie entirely outside the areas that need to be displayed. If there's one rectangle in the array, only subviews that lie partially or wholly within the rectangle will be displayed. If there are three rectangles, only subviews that fall partially or wholly within the second or third rectangles will be displayed. The display methods do nothing with the first rectangle in the array, the union of the other two.
- **display:::** constructs a clipping path around the rectangles if requested to do so in its third argument. This clipping is in addition (and prior) to the clipping done when focusing on the View. Only the View that receives the initial **display:::** message is clipped in this way; its subviews are not.

Before each subview is recursively displayed, the display methods intersect each of the rectangles with the subview's frame rectangle and translate the results to the subview's coordinate system. Each View that's displayed is guaranteed to be passed drawing rectangles that:

- Are in its own coordinate system.
- Designate no areas outside of its visible rectangle. The drawing rectangles may designate an area to draw in that's smaller than the visible rectangle, however.

See "Drawing Rectangles" above for information on how to use these rectangles in a **drawSelf:::** method.

Displaying Background Views

When a View surrenders a portion of a window—either because it becomes smaller or because it moves to a new location—it isn’t enough that it just display itself again. Any Views that it covered in the areas it abandoned must also be given a chance to redisplay themselves. A **display:::** message to its superview gives the superview and any sibling subviews this chance:

```
[superview display:&myOldFrame :1 :NO];
```

The message above is appropriate for a View that shrank. If it had moved, it would have passed an array of its old and new frame rectangles (and their union).

For a similar reason, when a View draws against a background provided by another View, it’s not enough that it send just itself a display message. The View that provides the background must also be given the opportunity to redisplay itself. In fact, any View that doesn’t erase or redraw everything within the drawing rectangles should make sure that the Views behind it are redisplayed. Instead of sending itself a **display:::** message, it should use the **displayFromOpaqueAncestor:::** method:

```
[self displayFromOpaqueAncestor:&bounds :1 :NO];
```

The three arguments to **displayFromOpaqueAncestor:::** are identical to the three arguments to **display:::**. The first is an array of rectangles specifying the area to be redisplayed; the second is the number of rectangles in the array. The third indicates whether to construct a clipping path around those rectangles.

The **displayFromOpaqueAncestor:::** method treats its arguments a bit differently than **display:::**, however. It first searches up the view hierarchy for the nearest ancestor View that guarantees that it will cover all the pixels within its frame rectangle with a fresh coat of opaque paint. **displayFromOpaqueAncestor:::** then translates the drawing rectangles to this View’s coordinate system and sends it a **display:::** message. If no View lower in the hierarchy meets the requirement, the **display:::** message is sent to the frame view at the root of the view hierarchy (the content view’s superview). The frame view erases the content area to the Window’s background color.

If the receiving View itself guarantees that it draws all its pixels in opaque paint, **displayFromOpaqueAncestor:::** is no different than **display:::**.

This procedure ensures that the areas that need to be updated are completely redrawn. The View that received the **displayFromOpaqueAncestor:::** message will get a message to display itself after the Views behind it have been displayed.

Registering as an Opaque View

If a View paints all the pixels within its frame rectangle with opaque paint, it should be registered as an opaque View:

```
[myView setOpaque:YES];
```

A NO argument to **setOpaque:** would unregister the View. The **isOpaque** method returns whether or not the receiver is currently registered:

```
BOOL erases;
erases = [myView isOpaque];
```

Once it's registered as opaque, the View becomes a potential recipient of a display message from **displayFromOpaqueAncestor:::**

Views are generally opaque because they erase before drawing, perhaps using the **NXRectFill()** or **NXEraseRect()** functions to fill their entire frame rectangles with a solid color before drawing detailed images. Box, ScrollView, and the Window's frame view are the principal opaque Views. A Text object becomes an opaque View after it receives a **setOpaque:** message with YES as the argument.

By displaying itself through the **displayFromOpaqueAncestor:::** method, a View can rely on other Views to erase for it.

Finding the Background View

The **opaqueAncestor** method returns the receiver, if it's opaque, or the nearest opaque View above it in the view hierarchy:

```
id backgroundView;
backgroundView = [myView opaqueAncestor];
```

In other words, **opaqueAncestor** returns the View that **displayFromOpaqueAncestor:::** would send a **display:::** message to. You can bypass **displayFromOpaqueAncestor:::** by sending your own display messages to the opaque View.

opaqueAncestor may return the frame view:

```
BOOL isFrameView;
isFrameView = ![[self opaqueAncestor] superview];
```

While you can send the frame view display messages, you should be careful not to alter it in any way.

Window's Display Methods

In addition to the display methods defined in the View class, Window has two display methods:

```
[myWindow display];  
[myWindow displayIfNeeded];
```

Both methods pass display messages on to the Views in the Window's view hierarchy. The **display** method displays all the Views in the view hierarchy, from the frame view on down. It's usually used to set up the Window's initial display. The **displayIfNeeded** method displays just those Views that need to be redisplayed. It's discussed under "Managing a Window's Display" below.

Automatic Display Messages

An application can send display messages to its Views at any time, but display messages are also initiated automatically, either as the result of a user action or as a corollary to some change in the View.

- After it has been resized by the user, a Window sends a **display** message to its frame view. Every View in the view hierarchy will redisplay itself to fit the window's new size.
- When a Window receives a window-exposed subevent (of the kit-defined event), it sends a **display::** message to its frame view. The message specifies a single rectangle enclosing just the areas of the Window that need to be redrawn.
- When a View is scrolled, the Application Kit copies the portions of the display that remain in view to their new locations. It then sends the View a **display::** message to have it redraw the update areas (as illustrated above in Figure 7-13, "Update Rectangles"). If the user scrolls the contents of a View vertically or horizontally, there will be a single area to be updated. If the user scrolls the View at an angle, there will be two update rectangles in the array.
- When a View attribute changes, the View may send itself a **display** message to make the change visible. See "Updating Views" in the section below.
- Whenever a Window is updated automatically, it may respond by sending its Views display messages. See "Updating Windows" below.
- When a Window receives a **display** message, every View in its hierarchy is displayed.

Because a View must be prepared to redisplay itself at any time in response to automatic display messages, all the code necessary to completely redraw the View should be contained in (or called by) its **drawSelf::** method.

Managing a Window's Display

The Window and View classes define some methods to help applications keep their on-screen displays current and make more efficient use of the display mechanism. They include:

- Methods for updating a Window automatically
- A convention for automatically displaying Views when they change, and methods for temporarily suspending the automatic display
- A way of temporarily suspending the display mechanism within a Window
- A way of temporarily suspending the part of the display mechanism that automatically flushes a window buffer to the screen

The sections below discuss each of these topics.

Updating Windows

Just before an off-screen Window is moved on-screen, it's sent an **update** message so that it can bring its display up-to-date with the current state of the application.

Window's default version of the **update** method simply returns **self**. A subclass can implement its own version to keep its display current. Menus use **update** messages to modify commands (from "Undo" to "Redo," for example), and to disable and reenable them, as appropriate.

On-Screen Windows

The automatic **update** message lets an off-screen Window alter its display before becoming visible. But Windows that are already visible also need to be updated periodically; the main menu, for example, is always on-screen. The Application object's **updateWindows** method fills this need by sending every on-screen Window in the window list an **update** message.

```
[NXApp updateWindows];
```

You can send NXApp an **updateWindows** message at appropriate times for your application, or you can arrange to have it performed automatically after every event:

```
[NXApp setAutoupdate:YES];
```

If NXApp receives the message above, it sends each visible Window an **update** message after each event has been processed in the main event loop or in a modal event loop for an attention panel. A NO argument to **setAutoupdate:** stops the flow of automatic **update** messages to on-screen Windows. (Off-screen Windows continue to receive them.)

Note that automatic **update** messages are sent only after NXApp dispatches an event from the event loops set up by **run** and **runModalFor:** messages. If an application sets up a modal event loop in response to an event, no **update** messages will be sent until the modal loop ends. If the modal loop is for an attention panel, no **update** message will be sent until after the first event is dispatched in the loop. Therefore, this mechanism can't be used to disable menu items upon entering the mode.

Updating Menus

For a Menu to respond to **update** messages, at least one of its MenuCells must be assigned a method that can determine how the MenuCell should be displayed. The **setUpDateAction:** method makes the assignment:

```
[menuItem setUpDateAction:@selector(fixMe:)];
```

The **updateAction** method returns the method selector that was assigned:

```
SEL theMethod;
theMethod = [menuItem updateAction];
```

A different method can be assigned to each MenuCell. The method should take just one argument, the **id** of the MenuCell, and it should be implemented by the Menu's delegate. (Menus inherit the delegate defined in the Window class.) The method's job is to check the current state of the application and alter the MenuCell accordingly. It should return boolean YES if the MenuCell needs to be redisplayed, and NO if it doesn't.

Whenever a Menu receives an **update** message, it has its delegate perform the updating methods for each of its MenuCells. If a method has been assigned to more than one MenuCell, it will perform once for each MenuCell it's assigned to. There is no default updating method; you must assign one with the **setUpDateAction:** method and implement the method in the Menu's delegate.

You can temporarily suspend updating for a particular Menu, while leaving it in place for all other Windows, by sending the Menu a **setAutoupdate:** message with NO as the argument:

```
[myMenu setAutoupdate:NO];
```

This disables the Menu's **update** method. By default, updating is enabled if an updating method has been assigned to any of the Menu's MenuCells; **setUpDateAction:** sends a **setAutoupdate:** message to the Menu with YES as its argument, thus automatically enabling the updating mechanism.

Updating Views

In general, when a View changes its state, it should display the change immediately. For example:

```
- setTitle:(char *)aString
{
    title = aString;
    [self display];
    return self;
}
```

There may be times, however, when it's best to postpone updating the display. For example, if several View attributes change at once, you probably want to wait until the last change is made to redisplay the View, rather than redisplay it after each change.

To temporarily prevent a View from being automatically displayed, you can send it a **setAutodisplay:** message with NO as its argument:

```
[myView setAutodisplay:NO];
```

The same message with a YES argument reinstates automatic displaying. By default, all Views are created with the automatic display flag turned on. The **isAutodisplay** method returns the current state of the flag:

```
BOOL doesDisplay;
doesDisplay = [myView isAutodisplay];
```

For this scheme to work, methods that alter View attributes must check whether it's OK to display the View. The **setTitle:** method shown above would need to be implemented more like this:

```
- setTitle:(char *)aString
{
    title = aString;
    if ( [self isAutodisplay] )
        [self display];
    return self;
}
```

If a method changes a View but doesn't redisplay it because the automatic display flag is off, it can set another flag (**vFlags.needsDisplay**) indicating that the View still needs to be displayed:

```
[myView setNeedsDisplay:YES];
```

This method works only while the View can't be automatically displayed (while the **isAutodisplay** method returns NO).

Other methods can check this flag to see whether they should display the View:

```
BOOL outOfDate;
outOfDate = [myView needsDisplay];
```

One method that checks the flag is **setAutodisplay:**. When it turns automatic displaying back on (when its argument is YES), it checks whether the receiving View needs to be displayed. If it does, **setAutodisplay:** sends the View a display message. Displaying the View clears its **needsDisplay** flag; thereafter the **needsDisplay** method returns NO.

Another method that checks the flag is **displayIfNeeded**. It works its way down the view hierarchy, displaying only those Views that have their **needsDisplay** flags turned on. For more on this method, see “Displaying If Needed” later in this section.

With this refinement, the **setTitle:** method illustrated above would look more like this:

```
- setTitle:(char *)aString
{
    title = aString;
    if ( [self isAutodisplay] )
        [self display];
    else
        [self setNeedsDisplay:YES];
    return self;
}
```

For convenience, the conditional statements in the **setTitle:** method illustrated above have been segregated into a separate View method, **update**. This last version of **setTitle:** can be simplified to three lines:

```
- setTitle:(char *)aString
{
    title = aString;
    [self update];
    return self;
}
```

It’s recommended that all Views use the **update** method and follow the conventions it depends on.

Suspending Display

Just as it’s sometimes a good idea to suspend the automatic updating of Views, it can also be a good idea to suspend all the displaying done within a Window for a short period of time. Suppose, for example, that several Views are changing and need to be redisplayed. You want the changes to be displayed all at once, not in piecemeal fashion. Rather than turn off the automatic display feature of each View, it’s more convenient to suspend the display mechanism for the entire Window.

Similarly, if an off-screen Window is undergoing a series of changes, with some of the changes cancelling others, you may want to wait until the Window receives an **update** message (just before it's placed on-screen) to display the changes. This could save on the volume of PostScript code sent to the Window Server.

The **disableDisplay** method suspends displaying within a Window:

```
[myWindow disableDisplay];
```

This message prevents the display methods defined in the View class from displaying any View within the Window. It should always be balanced with a **reenableDisplay** message when it's again OK to display the Window's Views.

```
[myWindow reenableDisplay];
```

Because there can be many reasons to suspend the display methods, pairs of **disableDisplay** and **reenableDisplay** messages can be nested. Displaying isn't reinstated until the last **reenableDisplay** message is sent.

While displaying is disabled, display messages that reach any of the Window's Views have no effect except to set the View's **needsDisplay** flag. This makes it easy to find the Views that couldn't be displayed and to redisplay them when it's again possible.

The **isDisplayEnabled** method returns whether or not displaying is currently suspended for the receiving Window:

```
BOOL canDraw;  
[myWindow isDisplayEnabled];
```

All the display methods defined in the View class and Window class are disabled by **disableDisplay**, except one. The **display** method defined in the Window class reenables displaying before sending a display message to its frame view.

Suspending flushWindow

The **disableDisplay** method described above prevents the display methods from sending any drawing code to the Window Server. On occasion, you may want a less severe suspension of the display mechanism. It might be more efficient to continue rendering images in the window's backup buffer, but wait to have them flushed to the screen.

For those occasions, **disableFlushWindow** and **reenableFlushWindow** can be used instead of **disableDisplay** and **reenableDisplay**.

```
[myWindow disableFlushWindow];  
/* code that displays Views within the Window goes here */  
[myWindow reenableFlushWindow];
```

disableFlushWindow prevents Window's **flushWindow** method from flushing the backup buffer of the receiving Window. After reenabling the method, the window buffer should be explicitly flushed:

```
[myWindow flushWindow];
```

Since **flushWindow** is used by the display methods, disabling it lets a number of images accumulate in the buffer before they're shown to the user.

disableFlushWindow works only if the receiving Window object manages a buffered window. Of the three buffering types (nonretained, retained, and buffered), only buffered windows require drawing to be flushed to the screen.

Like **disableDisplay** and **reenableDisplay**, pairs of **disableFlushWindow** and **reenableFlushWindow** messages can be nested. Window's **display** method doesn't automatically reenable flushing, however.

Displaying If Needed

A View's **needsDisplay** flag is set automatically when:

- An **update** message is unable to display the View (because automatic displaying is disabled).
- A display message is unable to display the View because displaying has been disabled for the Window where the View is located.
- A display message is sent to the View when the View isn't associated with a window. A View isn't associated with a window if it hasn't been assigned to a view hierarchy or if the Window Server hasn't yet created a window for the Window object.

In each case, the flag is a signal that there has been an unsuccessful attempt to display the View, which likely means that the appearance of the View doesn't reflect its current state. You can also set the flag directly using the **setNeedsDisplay:** method.

```
[myView setNeedsDisplay:YES];
```

The View class defines a **displayIfNeeded** method that displays a View only if its **needsDisplay** flag is on. It's thus a much more efficient way of choosing which Views to update than the other display methods.

The Window class also defines a **displayIfNeeded** method, which simply passes the **displayIfNeeded** message on to the Window's frame view. As the message works its way down the view hierarchy, only flagged Views are displayed. Displaying the View turns the flag off.

If a number of changes need to be made within a window, you can disable the display mechanism, make the changes, reenable the display mechanism, and then send the Window a **displayIfNeeded** message:

```
[myWindow disableDisplay];
/* make whatever changes are needed */
[myWindow reenableDisplay];
[myWindow displayIfNeeded];
```

If the methods that make the changes include automatic **update** messages, every altered View will be flagged and only those Views will be redisplayed. If **update** messages aren't sent automatically, you can send them yourself, or flag the Views directly with the **setNeedsDisplay:** method.

Modifying the Frame Rectangle

A View's initial frame rectangle is set by the class method that creates it. Slider's **newFrame:** method is an example:

```
id      mySlider;
NXRect rect;

NXSetRect(&rect, 20.0, 300.0, 15.0, 150.0);
mySlider = [Slider newFrame:&rect];
```

After a View is created, its frame rectangle can be relocated and resized by methods that insert new values in the View's **frame** instance variable:

```
[myView setFrame:&newRect];
[myView moveTo:40.0 :100.0];
[myView sizeTo:30.0 :200.0];
```

Two other methods alter the current values by a specified amount:

```
[myView moveBy:1.0 :1.0];
[myView sizeBy:-5.0 :10.0];
```

These two methods simply add the values recorded in the **frame** instance variable to the values they're passed and perform the **sizeTo:** and **moveTo:** methods to set the new values.

The **getFrame:** method provides the View's current frame rectangle:

```
NXRect rect;
[myView getFrame:&rect];
```

Views can also be rotated around the point recorded in **frame.origin**. Here a View is first rotated counterclockwise 108 from its superview's coordinates, then turned back 36 clockwise:

```
[myView rotateTo:108.0];
[myView rotateBy:-36.0];
```

Rotation turns the whole frame rectangle so that its sides are no longer aligned with its superview's coordinate system. This was illustrated earlier in Figures 6-7, "View Frame Rotation," and 6-8, "Default Coordinates."

Note: The **rotateTo:** and **rotateBy:** methods rotate the View itself, not its default coordinate system. In contrast, the **rotate:** method described earlier under "Modifying Default Coordinates" rotates the View's coordinate system, but not the View.

The **frameAngle** method returns the angle between the x- and y-axes of the superview's coordinate system and the sides of the frame rectangle:

```
float rotation;
rotation = [myView frameAngle];
```

If **frameAngle** returns 0.0, the View isn't rotated from its superview.

Resizing Subviews

When a Window is resized, the Application Kit automatically resizes the frame view and content view to fit the new dimensions of the window. You should never resize these Views yourself.

When a View is resized, especially if it's the content view, it may be necessary to adjust the size or position of its subviews.

An application could explicitly adjust all Views with newly altered superviews whenever it resizes a Window or View. It could do the same when it receives a **windowDidResize:** message indicating that the window, and therefore the content view, has been resized.

The Application Kit provides a simpler solution, however. You can specify how you want a subview to adjust to a resized superview and let the adjustment happen automatically.

A superview will automatically adjust its subviews if it's sent a **setAutoresizingMask:** message with YES as the argument:

```
[parentView setAutoresizingMask:YES];
```

Each subview must be told how to adapt with a **setAutosizing:** message:

```
[myView setAutosizing:(NX_WIDTHSIZABLE | NX_HEIGHTSIZABLE)];
```

The argument is a mask formed from the constants illustrated in Figure 7-14 below.

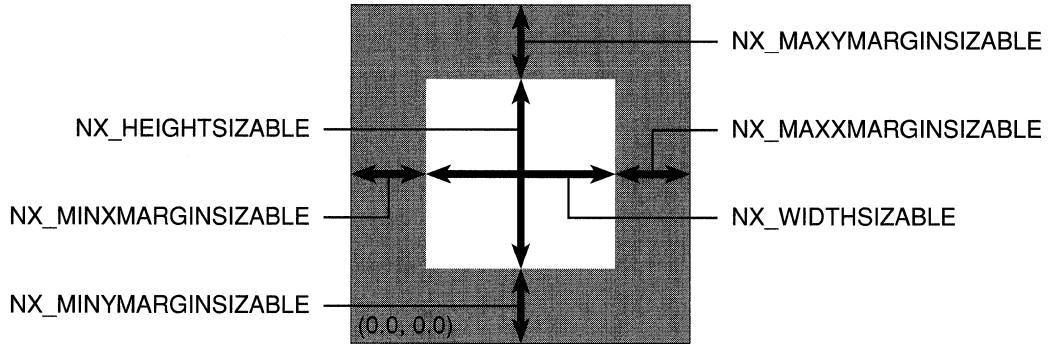


Figure 7-14. Resizing Constants

The constants specify what should be made to shrink or grow, both horizontally and vertically, to compensate for changes in the superview. In each direction, there are three choices:

- Resize the subview itself (`NX_WIDTHSIZABLE` and `NX_HEIGHTSIZABLE`).
- Resize the margin separating the edge of the superview from the sides of the subview with the lowest coordinate values (`NX_MINXMARGIN` and `NX_MINY`).
- Resize the margin separating the edge of the superview from the sides of the subview with the highest coordinate values (`NX_MAXXMARGIN` and `NX_MAXY`).

The effect of different masks can be illustrated if a few changes are made to the Little demonstration program listed under “Setting Up Event-Handling Objects” earlier in this chapter. First, the Window is given a resize bar and enlarged a little, and the Text object is centered in the content view so that it’s bordered by a uniform gray margin between it and the edge of the Window. This is shown in the first pane of Figure 7-15 below. Next, the content view is sent a `setAutoresizingMask:` message and the Text object is sent the `setAutosizing:` message illustrated above with a mask formed from `NX_WIDTHSIZABLE` and `NX_HEIGHTSIZABLE`. This tells the subview to resize itself while keeping its margins constant, as shown in the second pane of Figure 7-15.

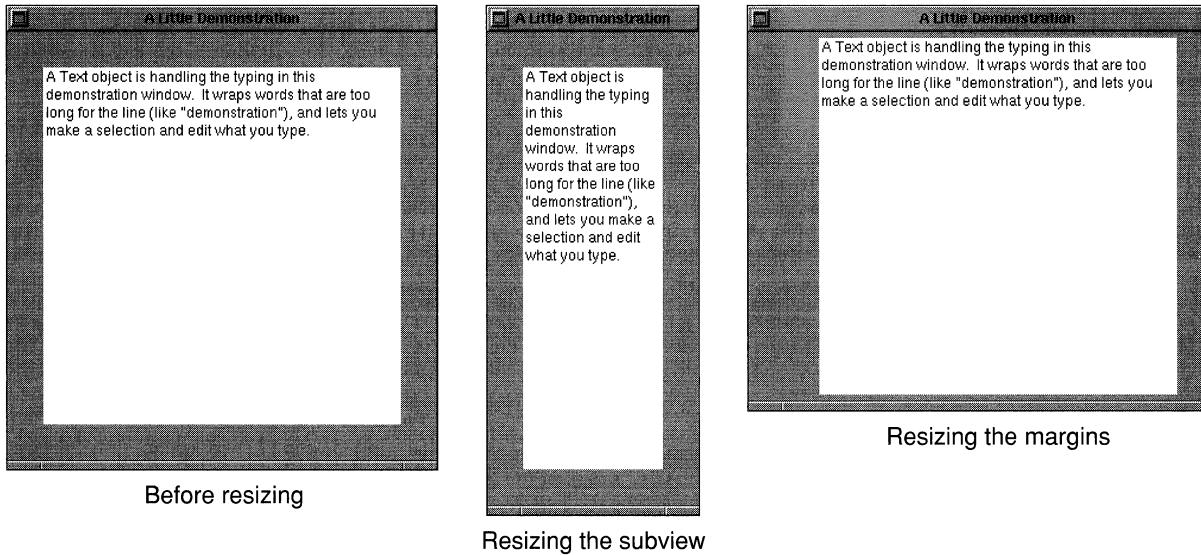


Figure 7-15. Resizing Subviews

It's also possible to resize the margins but keep the subview itself constant:

```
[myView setAutosizing:(NX_MINXMARGINSIZEABLE
| NX_MINYMRGINSIZEABLE
| NX_MAXXMARGINSIZEABLE
| NX_MAXYMRGINSIZEABLE)];
```

The result is illustrated in the third pane of Figure 7-15.

Other combinations are also possible, including resizing both the margins and the subviews proportionally. Each mask should specify something to resize in both the horizontal and the vertical directions. If not, the margins with the greatest coordinate values will be the ones that change.

The method that actually does the resizing is **superviewSizeChanged:**. You can override it for a View that needs to be resized in a special way. The argument passed to **superviewSizeChanged:** is a pointer to an NXSize structure containing the old size of the superview. The receiving subview can get the superview's new size directly from the superview by sending it a **getFrame:** message.

Printing

There are many different software components involved in printing with the NeXT computer. Printing begins in an application using the Application Kit connected to the Window Server. In response to a user's menu choice, the application, in conjunction with the Application Kit, generates the PostScript code required for rendering the images to be printed and sends it to the printing daemon, **npd**. **npd** establishes its own connection to the Window Server, prepares that Display PostScript context for imaging to the printer, and sends the Server the PostScript code, page by page.

The application's role in this chain of events is to work with the Application Kit to generate correct PostScript code which can be spooled to the printer via **npd**.

Generating PostScript Code

One of the architectural features of the NeXT computer is that it uses PostScript to draw on the display as well as the printer. This single imaging model simplifies printing, since if applications can draw their images on the screen, they are immediately capable of drawing them on the printer too.

When an application prints, it's usually printing all or part of a certain View, or possibly an entire window. One common case is for a document-oriented application to want to print the View containing the document in the key window. A View is printed by sending it a **printPSCode:** message, similar to how a View is displayed by sending it a **display** message. In fact, the printing machinery sends one or more display messages for various parts of the View. It's therefore essential that a View be able to successfully regenerate its image when sent a display message. The code to do this should be located in the **drawSelf::** method that's performed in response to display messages. If this criterion is met, basic printing should be fairly easy for any application to achieve, and more advanced applications should have sufficient hooks to tune the default printing process.

Although both the display and printer are driven by PostScript code, there are a few subtle differences between the PostScript code used to draw on the screen and the code generated while printing. One difference is that the PostScript code normally sent to the Window Server is an unbroken stream of PostScript commands, whereas the file of PostScript code created when printing "conforms" to a set of "Document Structuring Conventions." Most of these conventions involve special comments inserted into the PostScript stream. An important convention that the files created by the Application Kit follow is that the PostScript code is divided into a "prologue" and a "script" section. The prologue contains definitions used by the document, but no imaging code, and the script contains order-independent pages marked by various comments. The Application Kit provides a convenient framework for generating conforming files. (For more information, see the *Adobe Systems Document Structuring Conventions*.)

Although the application has full access to its state when generating the PostScript code, when that file is actually imaged on the printer, the application will be completely uninvolved. Here "state" includes any state the application has created for itself within its

PostScript execution context within the Window Server, or any state it has stored in PostScript shared VM in the Server. Also, the code may be printed via a network on a completely different machine than the one it was created on, with a completely different file system. For these reasons, the PostScript file that an application generates should not rely on any definitions outside itself, or any other state that cannot be guaranteed when the file is actually interpreted. For example, it isn't possible in the printing code to composite bits from an off-screen window that was created earlier in the application, since that window will not exist when the file is actually imaged.

It should be possible to print a file generated with the Application Kit on a PostScript device other than a NeXT computer and 400 dpi Laser Printer. To ensure this compatibility, you should be careful not to use Display PostScript extensions to draw while printing. This includes compositing operators, operators referring to windows or events, instance drawing operators, and graphics state operators. A few operators will be predefined in the template prologue generated by the Application Kit, and will thus be usable during printing. It's certain that **rectclip**, **rectstroke**, and **rectfill** will be simulated exactly in this prologue, and that the operators listed below will be defined to be a NULL operations.

flushgraphics	hidecursor
execuserobject	revealcursor
defineuserobject	obscurecursor
setcursor	showcursor

Other operators may be predefined in the future. Applications may use the global variable NXDrawingStatus (described in the next section) to determine if they are displaying to the screen or printing.

Another important difference between displaying on the screen and printing is the data format. Almost all communication with the Window Server happens in an efficient, binary format, facilitated by the **pswrap** tool and the **dpsclient** library. When printing, ASCII PostScript code is written to a disk file. Fortunately, most applications will be unaffected by this format difference, since the Application Kit does sufficient setup to make the **dpsclient** library convert outgoing PostScript code from binary to ASCII when printing. Therefore, the same **pswrap**-generated functions can be used for displaying and printing.

Application Kit Printing Architecture

The printing machinery in the Application Kit is structured to take advantage of the object-oriented environment. All Views and Windows inherit a method named **printPSCode:**, which should do a reasonable default job of generating a PostScript file for that View or Window. Almost all of what **printPSCode:** does is call other support methods in View. Applications modify the printing behavior of that View by overriding these other methods. Since the various functions of printing are broken out into these other separate methods, it's easy to change any aspect of printing by overriding a simple method. When overriding many of these methods, an application will be able to affect the desired change to the default behavior, and still call upon the View superclass for most of the method's implementation.

For example, let's say you want to insert the title of the document your application is printing into the conforming PostScript file as a “%%Title” comment. You also want to add some conforming comments to the spooled file's header. As your View generates its prologue, it calls a method whose charter is to generate the start of the prologue. You could override this method as follows:

```
- beginPrologueBBox: (NXRect *)boundingBox
    creationDate: (char *)dateCreated
    createdBy: (char *)anApplication
    fonts: (char *)fontNames
    forWhom: (char *)user
    pages: (int)numPages
    title: (char *)aTitle
{
    [super beginPrologueBBox:boundingBox
        creationDate:dateCreated
        createdBy:anApplication
        fonts:fontNames
        forWhom:user
        pages:numPages
        title:<title of current window's document>];
    DPSPrintf( "%%%SomeComment: %d", someNumber );
    return self;
}
```

During printing, various parameters control aspects of the printing job, such as how many pages are printed, and the size of the paper being printed on. This information can be set by the user through the PageLayout and Print panels, and is stored in an object of class PrintInfo. While a given print command is being executed, this information can be retrieved from a global PrintInfo kept by the application. For example, to find out the last page requested to print by the user, you could use the expression:

```
[ [NXApp printInfo] lastPage];
```

Simple applications will probably be content with having a single PrintInfo object that's created and initialized by the Application Kit. More advanced, document-oriented applications will probably want to store some pieces of the PrintInfo's data with their document, such as the paper type the document was created for. Such applications will need to ensure that the correct PrintInfo object is present in the Application object for all that ask for it. This could be done by either overriding Application's **printInfo** method and returning an appropriate object, or by setting the current PrintInfo object every time the active document changes.

One other useful piece of global information available to the application is the variable NXDrawingStatus, which can be NX_DRAWING, NX_PRINTING, or NX_COPYING. This variable reflects whether the program is generating PostScript code for the display, the printer, or the pasteboard. Sometimes applications will need to do some conditional drawing based on whether they're printing or not. NXDrawingStatus can be tested in a View's **drawSelf:** method to know the current drawing mode.

Pagination

When the Application Kit is printing, it loops through all the pages being printed, determines the portion of the View being printed that belongs on the current page, and tells the View to display that portion of itself. The goal of pagination is to determine what parts of the View should appear on which page.

Pagination happens in two modes. In the first mode, the Application Kit applies a recursive algorithm to the View being printed, which allows the View and its subviews to participate in how they are split up onto various pages. By default, a View tries to alter the pagination boundaries so that it's not split in two. Some Views will want to override the **adjustPageWidthNew:left:right:limit:** or **adjustPageHeightNew:top:bottom:limit:** method to change how they're broken up when they cross a page boundary. For example, Text uses the second method to make sure that lines of text are not cut in half when the Text object crosses a page boundary.

In the second mode of pagination, the application tells the Kit where the various pages lie. This mode will be used by advanced applications that do their own pagination, and know where their page breaks lie. In this mode, the Application Kit will ask the application for the rectangle for a given page with the **getRect:forPage:** method, and will later tell the View to display this same rectangle.

The first of these pagination modes is the default. It's fairly automatic, and should produce reasonable results for many applications. Views can override the **knowsPagesFirst:last:** method to tell the Application Kit that they will use the second method, and be responsible for their own pagination.

Image Placement on the Page

Pagination determines what rectangle of the View told to print will be drawn on a given page. Before the View is told to display that rectangle, the Application Kit must know how to place that image on the physical page.

For each page printed, the View told to print is sent the **placePrintRect:offset:** message. This method is passed the rectangle being printed in page coordinates, and is to return the offset from the lower left corner of the page that should be used to position the image. The area of the paper being used can be obtained from the PrintInfo object with the **paperRect** method.

The default implementation of **placePrintRect:offset:** uses some bits in the PrintInfo object to determine whether to center the image, or to align it with the top and left margins on the page.

Display PostScript Contexts

The Display PostScript system has facilities for controlling the concurrent execution of multiple execution contexts within the Window Server. The **dpsclient** library extends the concept of a Display PostScript context for clients to be any output channel to which the library sends PostScript code. During normal operation, the application has a single binary connection to the Window Server through which it gets all events and performs all drawing. This connection is represented by the context instance variable of the Application object. When printing is begun, the Application Kit creates a second DPSContext whose output is in ASCII format. This context is stored in the PrintInfo object during printing. During printing, this new context is made the current context, and hence the PostScript generated by the application is sent to that file, and not to the Window Server.

Sometimes an application needs to communicate briefly with the Window Server while generating PostScript code. For example, it may need to read some data from the Window Server as part of doing some drawing. In these cases, the normal server DPSContext can be made the current context temporarily. After communicating with the Window Server, the context should be reset back to the context used for printing:

```
DPSSetContext( [NXApp context] );
/* talk to the Window Server here */
DPSSetContext( [[NXApp printInfo] context] );
/* resume generating PostScript code to be printed */
```

Panels

There are three panels by which the user controls the various parameters of printing. The first is a PageLayout object. This panel is used to set properties that affect printing that affect how a WYSIWYG document is displayed on the screen. Applications will want to save most items from this panel with their documents.

The second panel is a PrintPanel object. The PrintPanel holds attributes of a given printing session that affect how the document is printed. Applications will not want to save these attributes with their documents.

The third panel is a ChoosePrinter object. It's invoked from the PrintPanel and permits the user to choose a printer to print on.

Each application has only one copy of each of these panels. If a panel has already been created, requests sent to the class object for a new panel will return the one already created.

Both of these panels load their contents from the global PrintInfo in the Application object when they come up, and save their values back to the same PrintInfo. The PrintInfo object is where to go for parameters for the current print job, not the various controls in the panels.

Applications will sometimes want to add extra controls and features to these panels specific to themselves. The controls should be contained in a View that's added to the panel with the **setAccessoryView:** method. You can define a subclass of the PrintInfo object to store

information set by the accessory View and redefine the panel's **writePrintInfo** method to put it there. To initialize the display in the accessory View with information stored in the PrintInfo object, you can redefine the panel's **readPrintInfo** method.

When using Interface Builder to create your application's interface, you can easily make use of the Application Kit's printing panels. For the PageLayout panel, you should create a menu item in your Window menu called "Page Layout...", and have it send a message to either your subclass of Application or a custom object of your own design. This object should then get the PageLayout panel, and run it. For example, if the menu item's action is a **doPageLayout:** method:

```
- doPageLayout:sender
{
    [[PageLayout new] runModal];
}
```

You'll rarely invoke the PrintPanel in such a direct manner, since it's run by the **printPSCode:** message. However, you still must determine which View or Window to send the **printPSCode:** message to when the user chooses the print command. You should put a "Print..." menu item in your main menu. If you have a simple application that always prints the same View or Window, you can set this item's target to be that object, and its action to be **printPSCode:**. If you have a more advanced application, you may need to send your subclass of Application (or a custom object) a message you invent. For example, you might set the menu item's action to be **doPrinting:** and implement the following method to print the window containing the active document:

```
- doPrinting:sender
{
    [[self mainWindow] printPSCode:sender];
}
```


Chapter 8

Interface Builder

8-6	Interface Builder and Program Design
8-7	The Interface File
8-8	The Interface File's Owner
8-8	The Project
8-9	The Process
8-11	Interface Builder Tutorial
8-11	Project One: An Interface Application
8-12	Getting Started
8-14	Creating a Project
8-16	Editing Objects
8-18	Adding Objects
8-21	Testing the Interface
8-21	Preparing to Compile the Application
8-22	Makefile
8-23	example1_main.m
8-24	Compiling the Application
8-25	Running the Application
8-25	Project Two: A One-Button Calculator
8-26	Creating the Interface
8-27	Defining the Calculator Class
8-30	Connecting the Objects
8-32	Writing the Calculator Class Definition Files
8-33	Calculator.h
8-33	Calculator.m
8-35	Compiling and Running the Application
8-35	Project Three: Modifying the Calculator
8-36	Adding a Submenu
8-37	Modifying Calculator.h
8-38	Modifying Calculator.m
8-40	Adding an Icon
8-42	Adding Sound
8-44	Project Four: A Text Editor Using Two Interface Files
8-45	The Text Editor's Design
8-47	Creating the Application's Interface
8-48	Defining the Distributor Class
8-48	Editing the Class Files
8-49	Connecting the Objects

8-50	Creating the Module's Interface
8-50	Defining the TextEditor Class
8-51	Editing the Class Files
8-52	Connecting the Objects
8-52	Compiling and Running
8-53	Interface Builder Reference
8-53	The Main Menu
8-53	Info
8-53	File
8-54	Edit
8-54	Font
8-54	Print
8-54	Windows
8-54	Layout
8-54	Hide
8-55	Quit
8-55	The File Menu
8-55	Open
8-56	New Application
8-56	New Module
8-57	Save
8-58	Save As
8-58	Save All
8-58	Hide File
8-58	Close File
8-58	Project
8-59	Make
8-60	Test Interface
8-60	The Edit Menu
8-61	Cut
8-61	Copy
8-61	Paste
8-62	Delete
8-62	Select All
8-62	The Font Menu
8-63	The Windows Menu
8-63	Palettes
8-63	Inspector
8-64	Page Layout
8-64	Close Window
8-64	The Layout Menu
8-65	Bring to Front
8-65	Send to Back
8-65	Size to Fit
8-65	Same Size
8-65	Group
8-66	Ungroup
8-66	Align

8-66	Resize Window
8-67	The Align Menu
8-67	Alignment
8-67	Make Row
8-68	Make Column
8-68	Set Grid On or Set Grid Off
8-68	Show Grid or Hide Grid
8-69	The File Window
8-70	File's Owner
8-71	First Responder
8-71	Font Manager
8-72	Icons
8-73	Sounds
8-73	Classes
8-76	The Palettes Window
8-77	The View Palette
8-78	The Window Palette
8-78	The Menu Palette
8-80	The Inspector Window
8-81	The Attributes Display
8-81	TextField Attributes
8-83	Button Attributes
8-85	Matrix Attributes
8-88	Form Attributes
8-89	FormCell Attributes
8-90	Box Attributes
8-91	Slider Attributes
8-91	ScrollView Attributes
8-93	CustomView Attributes
8-93	Window Attributes
8-95	MenuCell Attributes
8-96	File's Owner's Attributes
8-96	Icon Attributes
8-97	Sound Attributes
8-98	The Connections Display
8-99	The Autosizing Display
8-100	The Miscellaneous Display
8-101	The Class Display
8-101	The Project Display
8-101	The Class Inspector
8-102	The Project Inspector
8-103	The Attributes Display
8-106	The Files Display

Chapter 8

Interface Builder

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/ReleaseNotes/InterfaceBuilder.rtf](#)

Interface Builder speeds the creation of applications by letting you define an interface (and in some cases, an entire application) graphically rather than by writing Objective-C code. With Interface Builder, you manipulate graphic representations of Application Kit objects just as if you were using a graphics editor to create a drawing. However, you're not restricted to using Application Kit objects; Interface Builder lets you add objects of your own design. Once you've gathered and arranged the objects that will make up your application, Interface Builder lets you define the interactions among them. Even before you write a line of code, you can run your application's interface in simulation to check its operation.

This chapter is divided into three main parts: an overview, a tutorial, and a reference section. The overview introduces some of the basic concepts that you'll need to understand to get the most from the other parts. The tutorial guides you through several simple projects that familiarize you with how Interface Builder operates and with how building an application with Interface Builder differs from building one from scratch. Finally, the reference section explains each of the many tools and commands available in Interface Builder.

Even if you're new to this computing environment, you'll find that using Interface Builder, you'll be able to create a simple application with a minimum of time and effort. This efficiency results from working directly with the application's objects, rather than with files of programming code. However, the more you know about the Application Kit and the more comfortable you are with programming in the Objective-C language, the easier it will be for you to create more ambitious applications. For information on the Objective-C language, refer to Chapter 3, "Object-Oriented Programming and Objective-C." See Chapters 6 and 7, "Program Structure" and "Program Dynamics," for more information on programming with the Application Kit.

Interface Builder and Program Design

The Application Kit defines a library of user-interface objects that you can select from for your application. Interface Builder makes the selection process a graphical one: You simply drag the object you need from a palette in Interface Builder to a destination in the application you're building. By building an application in this way, you can be sure that its interface will work properly and will, in a broad sense, conform to the interface standards for the NeXT computer.

Once an object is added to your application, you can adjust the values of many of its instance variables directly. For example, to change the size of a button, you drag one or more of its sides to a new position. Changing the image on the screen changes the value of the Button object's **frame** instance variable. For attributes that aren't easily represented graphically, Interface Builder provides an Inspector window that lets you enter the values for selected instance variables. You set the maximum and minimum values of a slider with the Slider Inspector, for example.

Interface Builder also lets you interconnect objects so that they can communicate with one another. The connections are made through an object's outlets. An outlet is an instance variable of type **id** that you can identify with another object in the application. Common examples of outlets include a Control's **target** or an Application or Window object's **delegate**. When your application begins execution and its interface objects are created from the interface file, outlet variables are automatically initialized to the **ids** of the objects you've previously specified within Interface Builder. (See "Outlets" in Chapter 6 for more information.)

The objects in the Application Kit are general-purpose and fill the needs of a wide cross-section of applications. What makes your application unique is the code you write. For example, the Application Kit provides the Buttons and other Views you need to implement an interface for a calculator, but you have to create the computational engine.

Interface Builder and Objective-C encourage a style of programming that puts the unique code of your application in one or more objects of your own design. The application's user-interface objects handle routine business, such as displaying an Info panel or hiding the application, and also serve to interpret the user's actions for the objects you design. If the user clicks the calculator's Add button, the Application Kit highlights the button and then sends a message to your calculator object to perform the addition.

Using this style of programming, your application will generally contain a number of standard Application Kit objects and one or more subclasses of Object and View. Most often, the subclasses of Object embody the logic that's unique to your application, and the View subclasses contain the drawing code that's unique to your application. You'll rarely need to create subclasses of other Application Kit classes.

The Interface File

The interface specification you develop using Interface Builder is saved in an interface file. This file (identified by a “.nib” file extension) contains:

- Class interface information for Application Kit classes your application uses and for any subclasses of Object and View that you define. At run time, the Application Kit sends messages to create objects of each of these classes.
- Specifications for the Application Kit objects your program contains. This includes each object’s size, location, and position in the view hierarchy, along with other specifications. At run time, the Application Kit sends messages to initialize each of these objects.
- Information about how outlets should be initialized at run time.
- Information about action messages and their targets.
- Sound and icon data.
- A reference to an owner object. (The interface file’s owner is described in the next section.)

Each application you create with Interface Builder will have at least one interface file, the main interface file. This file contains the specifications for the application’s main menu and perhaps other objects. An application can have only one main interface file.

More complex applications may have other interface files in addition to the main interface file. For example, an application might have two interface files: one containing specifications for the main menu and other primary interface objects and the other containing specifications for the Windows, Buttons, and other objects of the application’s help system. If a user requests help, the help system’s objects are created using the information in the help system interface file. These new objects are connected to the structure of the application through their owner, an object that already exists within the application. By not being created unless they’re needed, the help system objects don’t consume system memory or add to the application’s startup time.

Whether your application has one or multiple interface files, when it’s compiled, its interface data is copied from the interface file or files and placed in a segment of the Mach-O format executable file. This eliminates the need for auxiliary files at run time. However, because the interface file becomes part of the executable file, you need to recompile the application whenever you make changes to the interface file. Fortunately, if the changes you make only affect the interface file, recompiling the application takes only a few moments.

The Interface File's Owner

The interface file's owner is an object that's external to the interface file and that's the conduit between objects in the interface file and the other objects in your application.

Each interface file has one, and only one, owner. For small applications, the owner is generally NXApp, the application object itself, although it can be an object of any class. The owner is the only external object that may be the explicit target of action messages from Controls within the interface file. The owner may also have outlets that will be initialized at run time to the ids of objects within the interface file.

The owner must exist before the interface objects are loaded. For example, for a simple application Interface Builder generates a main file that follows this sequence of messaging to create the owner, load the interface information, and then run:

```
NXApp = [Application new];
[NXApp loadNibSection:@"example1.nib" owner:NXApp];
[NXApp run];
```

The Project

In Interface Builder, each application is part of a project. A simple project consists of:

- A project directory. The directory contains all the files that you want to be part of the project. When you add a file that isn't in the project directory to the project, Interface Builder copies it from its current location into the project directory.
- A project file. This file, always called **IB.proj**, keeps the pieces of the project organized. It records the application's name, the names of the files that make up the project, and other information that's needed to create and update the application's makefile (described below).
- An interface file.

After you create a project, each time you save the application's interface file, Interface Builder also updates **IB.proj** with any changes you've made to the project.

Interface Builder uses the information in the project file to create three files that you'll need to compile the application:

- A makefile. The makefile, used by the **make** utility, specifies which files are required by the compiler and linker to build your application. (For more information on makefiles and **make**, the program that accesses the makefile, see the UNIX manual page for **make**.)
- A main file. The main program file contains the **main()** function, the entry point for your application.
- An icon header file. This file has the name of the application plus the extension “.iconheader”. It contains information which, when later incorporated into the executable file, the Workspace Manager uses to associate icons with the application and its documents.

More complex projects can include multiple interface files, custom class definition files, and so on. No matter how many files make up the project, there's only one project file that oversees them.

The Process

As shown in Figure 8-1, building an application with Interface Builder involves these major steps: creating an interface, creating a project, and compiling the application.

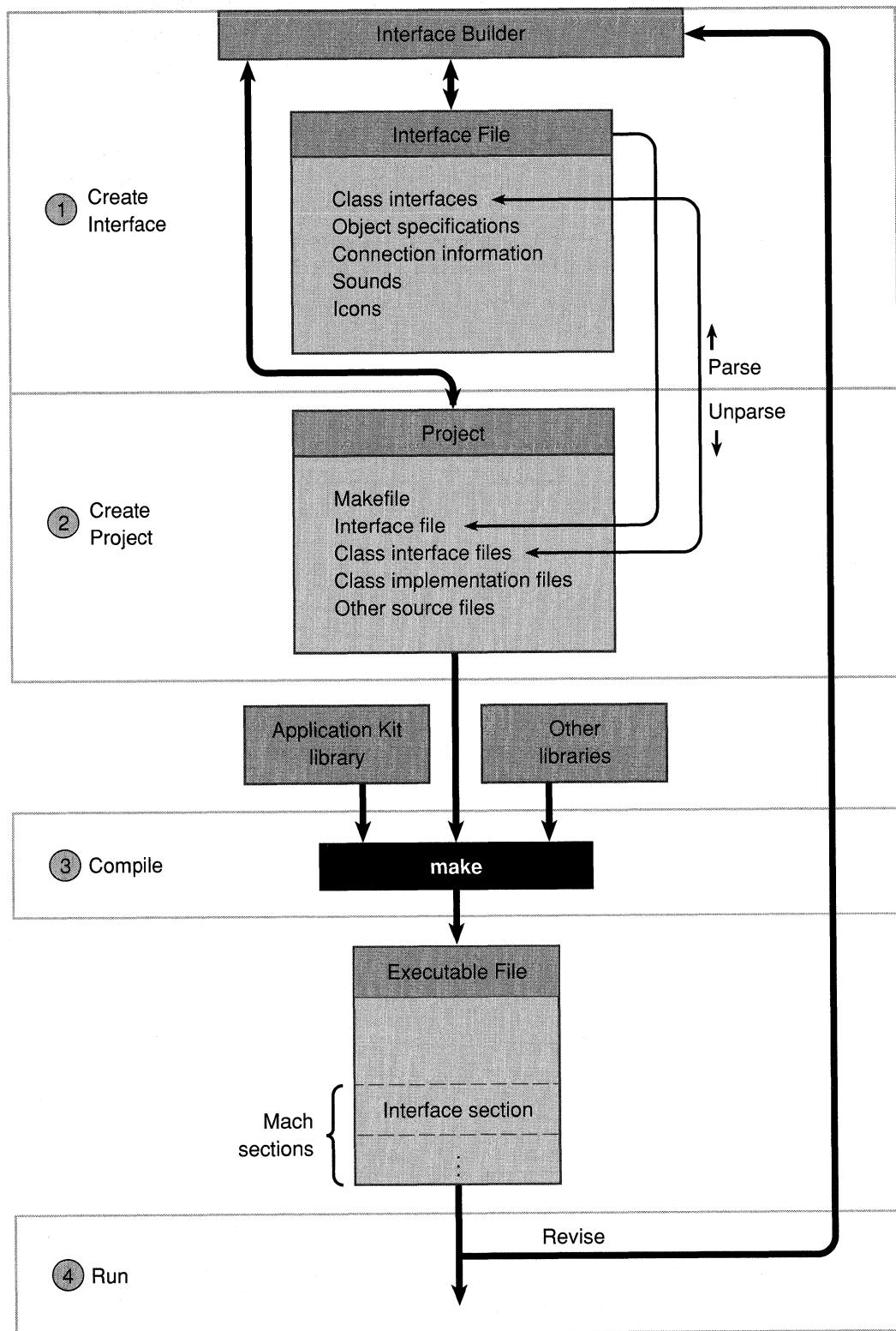


Figure 8-1. Building an Application

You build an application by first defining its interface and then saving the interface data in a file. The interface file, in turn, becomes part of a project, which will include a makefile and other source files. When all the pieces of the project have been brought together, you compile and link them using the **make** utility.

Notice that although a project comprises several separate files, most applications are represented by a single executable file after they're compiled. This is because Interface Builder's standard makefile instructs the linker to install the interface file (and possibly other source files) in segments of a Mach-O format executable file.

Notice too that the diagram includes several return pathways in the general flow from Interface Builder to executable file. For example, from your actions in Interface Builder, you build an interface file, but you could read in an existing interface file for further modification. Similarly, within Interface Builder you can define an Objective-C class. Interface Builder then "unparses" the description and writes the skeletal source files for the class. Alternatively, Interface Builder can "parse" an existing class definition file so that you can use the class within your project. And of course, once you've run the application, you can return to Interface Builder to revise any piece of its project before compiling and running it again. The projects described in the tutorial section that follows demonstrate how these various pathways aid in application development.

Interface Builder Tutorial

As a quick introduction to Interface Builder, this tutorial shows you how to build several simple applications. These projects are designed more to illuminate the features of Interface Builder and of the Application Kit than to create useful applications. In fact, the first project creates a minimal application, one consisting entirely of interface objects, with no objects of your own design. This project will help you understand what parts of the application development process can be done entirely within Interface Builder and what parts remain for you to do independently. The next two projects show you how to create a single-function calculator by using many of the pieces introduced in the first project and adding an object of your own. After building and running the calculator, you extend its features by adding a submenu of calculation types and by introducing icons and sound to the application. The final project shows you how to build a simple multi-windowed text editor.

Project One: An Interface Application

An application on the NeXT computer has at least three major components: a standard window, a main menu, and an Info panel. The Application Kit defines these objects, and Interface Builder helps you assemble them, along with other objects, into a program. This project shows you how to use Interface Builder to create a complete, though content-free, application using these building blocks. This will give you a better idea of which parts of the application development process can be done entirely in Interface Builder and which parts can't.

Start Interface Builder from the Workspace Manager, either from its location in **/NextApps/InterfaceBuilder** or from the dock, if its icon is there.

When the application starts, it displays its main menu and the Palettes window. Directly or indirectly, the main menu gives you access to all of Interface Builder's commands. The Palettes window holds images of the various Application Kit objects you might include in your program.

Getting Started

You start building an application in Interface Builder by creating a new interface file (step 1 in Figure 8-1 above). Choose New Application from the File submenu. Three windows appear: a menu titled “Untitled1,” a standard window titled “MyWindow,” and a window containing icons. The first two windows are part of the application you’re building; the third window, known as the File window, is an Interface Builder window that gives you access to the resources and top-level objects in your application’s interface file. “Untitled1” is the default name of the application (and thus its main menu) as well as the name for the interface file you’ll be building. The File window’s title bar displays this file name within the current directory. We’ll change these titles and the interface file’s location later.

The File window appears as in Figure 8-2.

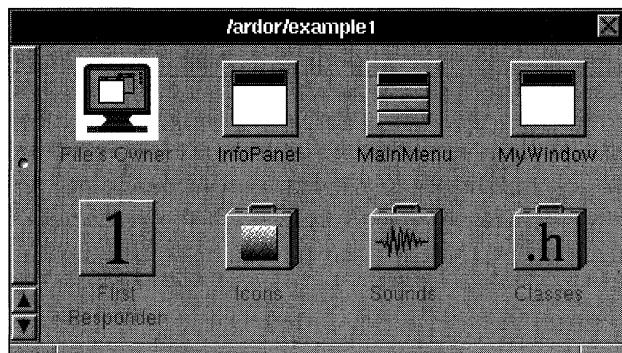


Figure 8-2. The File Window

The File window’s title displays the default name of the interface file. The icons within the window correspond to the objects and resources in this interface file. Object icons are squares; resource icons look like small suitcases.

In the upper left corner of the File window is an icon representing the interface file’s owner. The remaining icons in the top row represent the application’s top-level interface objects: “MainMenu” and “MyWindow” correspond to the first Menu and Window objects in the interface file. “InfoPanel” is the name of the Panel object used for the Info panel. The Info panel isn’t shown since it’s normally not visible unless the user chooses the Info command.

In the second row, the icon titled “First Responder” represents the object that is the first responder within MyWindow. (For more information about the use of the first responder, see “First Responder” in the reference section of this chapter.) The last three icons represent the various icon, sound, and class resources available to your application.

By double-clicking an object icon, you can select and display the corresponding object and any objects it contains. In this way, these top-level objects give you access to all the objects in your application. To see the Info panel and the objects that it contains, for example, double-click the InfoPanel icon. A generic Info panel appears (Figure 8-3). Later, we’ll customize the information in this panel.

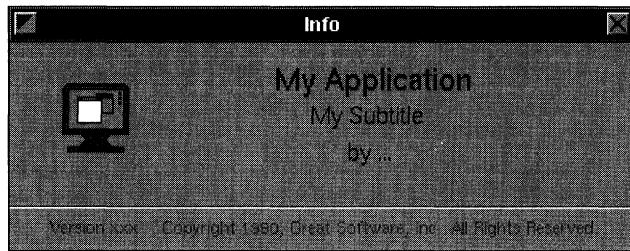


Figure 8-3. The Info Panel

If you select an object icon in the file window and then click the object’s name below the icon, you can edit the object’s name. Only object names that are displayed in black can be edited, however. Changing an object’s name has no effect on the object’s class; it only changes the name Interface Builder uses to keep track of the various objects within your application. (See “The File Window” in the reference section for more information.) For now, however, leave the default names.

Before going further, save the application interface you’ve created so far in a file called **example1**. So that you can make this file part of a project later, create the project directory **project1**. The fastest way to do this is to use the Save panel. Choose Save from the File menu. When the panel opens, enter:

```
project1/example1
```

Another panel opens and asks if you want to create this path. By clicking the Create button, you create the directory and write the file **example1.nib**. (Interface Builder adds the “.nib” extension to the name of the interface file if you don’t specify it.) Notice that the title of the File window changes to report the path and file name of the interface file. Also, the main menu’s title displays the name of the application, which Interface Builder takes to be the same as that of the interface file.

Creating a Project

Now that there's a project directory and a file to manage, let's create a project to manage it. (This is step 2 in Figure 8-1 above.) Choose the Project command in the File menu. This command opens the Inspector window for the project. The Project Inspector informs you that there's no project file in the current directory and asks whether you want to create one. Confirm that you do. Interface Builder creates the project file **IB.proj**, a makefile, and a main program file, and adds them to the project directory. It then shows the Files display for the project, as shown in Figure 8-4.

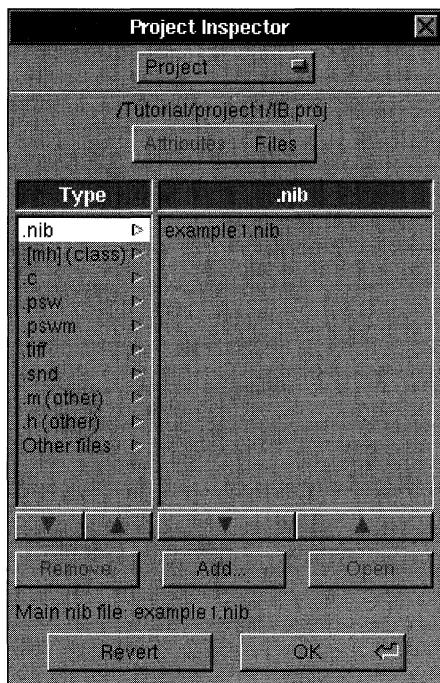


Figure 8-4. The Project Inspector

The Inspector window is a general-purpose editor. Most of the time its display is keyed to the object that's selected within the application. For example, if a button is selected in the application's window, the Button Inspector appears. However, since we opened the inspector using the Project command, the window shows the Project Inspector.

The Project Inspector gives you access to the information stored in the project file. The window is organized into two displays. By clicking the Attributes/Files button near the top of the window, you can switch from the display of the project's attributes to the display of its source files. Let's first look at the Files display.

The Project Inspector's Files display gives you an organized view of the files that make up your application. The left column lists the types of source files, and, for a given type, the right column displays the names of any files of that type that your project contains. Notice that **example1.nib** has been added under ".nib" files. If you have an interface file open

when you create a project—and the interface file contains a main menu—Interface Builder will automatically add the file as the application’s main interface file. The file name of the main interface file appears in the bottom portion of the Project Inspector window.

Besides the main interface file, your project contains the main program file **example1_main.m**; choose the type labeled “.m (other)” in the left column to see the listing of this file.

In addition to reporting which files are part of your project, the Project Inspector gives you access to these files. By double-clicking a file name in the browser, the appropriate application opens the file to let you edit its contents. Double-click **example1_main.m** in the browser to see how this works. However, leave the file unchanged for now.

Now click one of Interface Builder’s windows to make the Project Inspector appear again and switch to the Attributes display. Near the top of this display are two text fields. The top field displays the application name (which by default is the same as the title of the main menu), and the next field shows its installation directory. The installation directory is where the files of the finished application will eventually reside (by default, the **Apps** directory within your home directory). During the building and testing phase of your application, these files will be kept in the project directory.

The Icons section of this window lets you associate icons with the application. The button on the left titled “Application” holds the application’s icon (which, unless you reassign it, is the generic icon for executable files). The other two buttons are provided for icons that can be associated with the application’s documents. For example, the Workspace Manager associates a file with a “.nib” extension with Interface Builder and displays the Interface Builder document icon.

Although we won’t do it now, you can assign an icon to the application or one of its documents by clicking the appropriate button and then clicking Set. An Open panel appears and lets you specify a TIFF file for the selected button. After you click OK, the icon appears. (If the TIFF file isn’t in the project directory, Interface Builder asks if you want to copy it into the project directory.) If the icon is for a document, you would next enter the file extension for this document type in the Document Extension text field.

The radio buttons within the box titled Project Type let you choose the nature of the project: whether you will be building an independent application or a customized version of Interface Builder. For now, leave the Application button selected.

The Options group lets you set whether Interface Builder generates a new main program file and a new makefile whenever you make a change that would affect these files. By default it does, ensuring that these files are kept consistent with the rest of your project.

Warning: From now on, whenever you make changes to your application that affect the application’s main file or makefile, Interface Builder automatically updates these files. For example, if you specify a different main interface file (which affects the main file) or add source files (which affects the makefile), Interface Builder will update the affected file. Because of this automatic updating, it’s best not to customize either of these files. However, if you want to prevent either of these files from being overwritten, use the appropriate button in this group.

Editing Objects

Interface Builder lets you edit objects in two ways: by manipulating them directly and by using an inspector.

You can directly change a window's location on the screen and its size. To change a window's location, simply drag it to where you want it to appear when the application is run. Menus don't obey this system, however. No matter where you place the menu during development, when the application runs, the menu follows the NeXT user interface guidelines by appearing at the upper left corner of the screen (unless the user specifies a different location using the Preferences application). To change a window's size, you use one of two methods, based on whether the window will be resizable when the program is run. If it will be, resize it by using the resize bar as you normally would. If it won't be resizable, you have to make it temporarily resizable by clicking the resize button  in the title bar to temporarily display a resize bar. To experiment, try resizing the Info panel.

You can directly adjust the size and placement of objects in a window. For example, click the Info panel's text that reads "My Application." The field that displays the text becomes selected.

Selection is indicated by eight control points that appear around the object, in this case a text field. You can manipulate these points to change the object's shape and size. Dragging a corner control point adjusts the object's width and height simultaneously. Dragging a side control point adjusts only its width or height, depending on the point. To move the entire object, press the mouse button while the cursor is within the rectangular area delimited by the object's control points and drag—taking care not to drag one of the control points. You can constrain the object to move in only the vertical or horizontal directions by Command-dragging it. If you start Command-dragging in the vertical direction, for example, no horizontal motion is possible until you release the mouse button and begin dragging again.

When you select another object within the window, its control points appear, and the previously selected object's control points disappear. To select all the objects in a window, use the Select All command in the Edit menu. You can also select a group of objects in a window by "rubberbanding," dragging out a rectangular area that includes or intersects the objects.

Selected objects can be moved as a group by moving any one of them, and they can be cut, copied, or pasted by using the corresponding commands in the Edit menu. The Cut, Copy, and Paste commands work within a single window, between windows in the same project, and even between windows in different projects.

You can edit the text displayed by an object by double-clicking the text. Using normal selection techniques, you can extend the selection to include more characters. Change "My Application" to read "example 1". Similarly, replace the three dots at the end of "by ..." with your name. You might also revise the version number under the icon button to read "Version 001".

To edit an object's attributes that can't be easily manipulated graphically, Interface Builder provides an Inspector window for the particular object. This window lets you initialize the values of selected instance variables of an object. The Button Inspector, for example, lets you set the type and appearance of a button, among other things.

Let's look at the inspector for the application's standard window, which is currently titled "MyWindow". Select the window (by clicking anywhere within its boundaries or by double-clicking its icon in the File window). Next, press the button at the top of the Inspector window and drag to Attributes in the pop-up list that appears. Notice that the Window Inspector (as shown in Figure 8-5) lets you set the window's title, class, and other attributes.

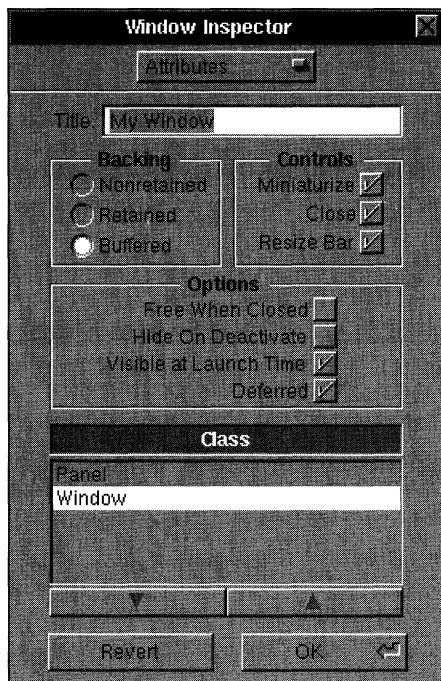


Figure 8-5. The Window Inspector

In later projects we'll explore the other options in more detail, but for now, change the window's title to "Test Window" or another title of your choice. Click the OK button to register the change.

Before going on, choose the Save command from the File menu to save the work you've done so far.

Adding Objects

Perhaps the easiest operation in Interface Builder is adding objects to an application: You simply drag the object from the Palettes window to the desired destination in your application. Before beginning, let's look at some of the features of the Palettes window, shown in Figure 8-6.

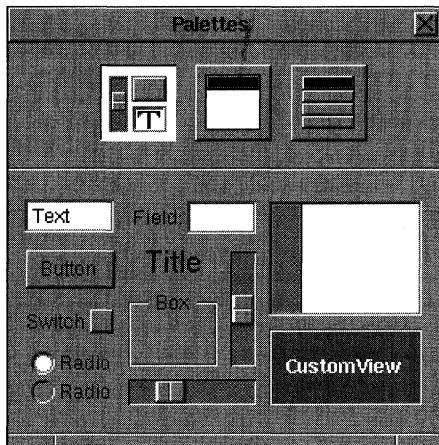


Figure 8-6. The Palettes Window

This window has three distinct displays, represented by the three buttons near the top; the left button gives you access to View objects, the middle to Window objects, and the right to Menus and MenuCells. When the Palettes window first appears, the left button is selected, and the lower part of the window displays the different View objects that the Application Kit provides for your application. You'll notice one object titled "CustomView." This is a placeholder for a subclass of View that your application defines, as will be discussed in more detail later.

Clicking the center button in the Palettes window displays the window palette. From this palette, you can add new windows and panels to your application. The last button displays the menu palette. This palette lets you add items to your application's menus. Some of these items are preconfigured for specific functions. For example, to add standard font manipulation commands to an application, you'd drag the Font menu item into the application's main menu. When the application runs, these commands will send the proper messages to a FontManager object to let the user modify the current font.

Let's add some objects to the application. First, click the View button in the Palettes window. Drag a Box object from the Palettes window into your application's standard window. You already know how to resize and position a selected object in a window; now try cutting, copying, and pasting this box.

To experiment with Interface Builder's drawing tools, arrange two boxes so that one partially covers the other, and then open the Layout menu. Select one box and then alternately select Bring to Front and Send to Back to see what these commands do. Next, choose Size to Fit. This command resizes an object so that it just accommodates its contents. Since there's nothing in the selected box, this command resets it to its minimum size.

Select two boxes (you could "rubberband" them or click one box and then hold down Shift while you click the other) and choose Same Size. The box you selected last is resized to match the size of the first box. Now, with both boxes selected, choose Group. This command has two effects: It visually groups selected objects by surrounding them with another box, and it makes the selected objects subviews of the surrounding box. Notice that if you move the surrounding box, the original boxes, being subviews, move with it whether or not they're selected. Remove the grouping by clicking Ungroup.

The Align command displays a menu of alignment tools. The first command, Alignment, opens a panel that affects how the other commands in the Align menu work. We'll take a closer look at this panel in a moment.

The Make Row and Make Column commands align a series of selected objects vertically or horizontally. If you select several objects and then click Make Row, the objects form a row to the right of the object that was nearest the left edge of the window. Similarly, clicking Make Column causes the objects to line up under the object that was nearest the top edge of the window. Try this with the two boxes.

The spacing between objects is determined by the original spacing between the two objects nearest the reference edge of the window. If these objects originally overlapped, the objects in the resulting row or column abut each other.

The Set Grid On command turns on an alignment feature in all your application's windows, making it easier to create pleasing layouts. Choose the Set Grid On command and then drag one of the boxes. Notice that the box moves in small increments both vertically and horizontally. Click Show Grid to make the alignment grid visible as a rectangular pattern of gray dots. You'll notice that when you move an object, the object's lower left corner jumps from dot to dot. The grid is only visible while you're building the application; it has no effect on your application's appearance in test mode or at run time. Both of these commands toggle, so a second click turns the feature off.

Now that you've seen the alignment tools, let's take a look at the Alignment panel.

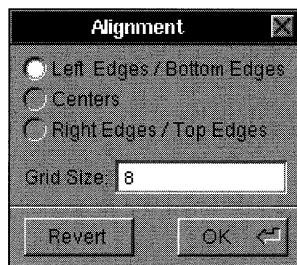


Figure 8-7. The Alignment Panel

The radio buttons let you set the part of an object's frame rectangle that's used as the reference point by the Make Column and Make Row commands. By default, objects are aligned according to their lower left corners. However, by clicking one of the other choices, you can align them according to their centers or their top right corners. Using the Grid Size field, you can set the width and height of the alignment grid. After changing these alignment settings, click OK to make the changes take effect. Until you click OK, you can always click Revert to reestablish the original settings. For now, we're done with the Alignment panel and Layout menu; you can close them.

Any of the Controls in the Views palette—in other words, the Slider, TextField, Form, and Button objects—can be made into matrices of objects by holding down Alternate while dragging one of the object's control points. For example, drag a Button into the window. While holding Alternate down, drag one of the corner control points diagonally across the window. When you've dragged the point far enough to make room for more Buttons, these objects appear. Try dragging the point vertically and then horizontally. In this way, you can make a row, column, or two-dimensional array of buttons. You can manipulate a Slider or TextField in the same way, but you can drag a Form only into a column. If you need a row or two-dimensional configuration of a Form object, you must create it programmatically.

The objects in a matrix act as a unit: Dragging one drags the entire matrix. To eliminate one or more objects from a matrix, hold down Alternate and resize the matrix so that the object or objects you want to eliminate fall outside the new limits of the matrix.

The spacing between objects in a matrix can be controlled by dragging a control point of a matrix while holding Command down. Experiment by dragging out a column of buttons and then stretching the matrix by holding down Command and dragging a control point.

To select one of the objects in a matrix, double-click the object. The object's highlighting indicates that it's selected. By double-clicking a second time, you can edit the text displayed in the object.

Editing the text in each of the objects in a matrix is made easier by the use of Tab to move from object to object. For example, edit the text in one button in the matrix of buttons. Press Tab, and you can immediately edit the text of the next button in the matrix. By repeatedly pressing Tab, you can access each of the objects in the matrix. Shift-Tab reverses the direction of motion so that the selection moves to the previous object.

Add examples of the other Application Kit objects from the View panel, but don't add a CustomView. The CustomView object is a proxy for a View subclass you write. By supplying this proxy, Interface Builder lets you specify the size, placement, and other parameters of a View subclass you'll supply. A later project will demonstrate the use of a CustomView object. Also for now, don't take anything from the window or menu palettes; you'll use these palettes in the later projects.

Note: Remember that you can remove an object from the application's window by selecting it and choosing the Cut command.

Testing the Interface

To test the application in simulation mode, choose the Test Interface command in the File menu. All of Interface Builder's windows disappear, leaving your application's windows on the screen. To indicate that it's in test mode, Interface Builder's application icon changes to display a large switch. Finally, your application's main menu moves to the upper left corner of the screen.

Your application's interface can now be tested. Even though it's running under Interface Builder, it should behave, with two small exceptions, as if it were a stand-alone program.

The exceptions affect the Hide and Quit menu commands. When your application is running in test mode, it doesn't display its application icon. Consequently, after you choose the Hide command, there's no way to recall your application's windows to the screen. To make your application's windows visible again, double-click Interface Builder's application icon. This restores your application to the screen and exits test mode. The Quit command, rather than quitting your application, exits test mode.

In all other respects, your application's interface operates normally. Buttons highlight when you click them, text in text fields can be edited, radio buttons work as you would expect, and the scrolling text view displays a scroll knob and buttons when you add sufficient text. In addition, the Info panel responds to the menu's Info command and its own close button.

In the normal course of application development, you'll probably pass through the build and test modes several times until you're sure your application's interface is perfect. After that, you'll write the code for any custom objects your application requires, compile the application, and then run it. In the next section, you'll see how to compile and run this sample application.

Before going on, choose the Save command from the File menu to save your work.

Preparing to Compile the Application

Before compiling the application, let's take a look at the pieces Interface Builder has provided. If you look in the project directory, you'll see these files:

- IB.proj
- example1.nib
- example1.iconheader
- Makefile
- example1_main.m

You may also see backup files for any of these files. A backup file is marked with a following tilde character (~) and contains the previous version of the file. For example, the backup file for **example1.nib** is **example1.nib~**.

IB.proj is the project file, which contains the information from the Project Inspector. The file **example1.nib** is the application's interface file, and **example1.iconheader** contains information about the application's icon. The information in these three files is in binary form and should only be edited indirectly, by using Interface Builder to change your project or its components.

Makefile, the file that coordinates the compilation process, is constructed from information in **IB.proj**. You generally make changes to the makefile indirectly, by changing the information in the Project Inspector. The last file, **example1_main.m**, is the main program file. This file contains the **main()** function, the entry point for execution. You may, on occasion, need to edit this file directly.

Let's take a closer look at **Makefile** and the main program file.

Makefile

Makefile controls the compilation and linking of the elements that make up your application. Interface Builder generates the makefile and fills in the names of your application's source files in the appropriate spots:

```
#  
# Generated by the NeXT Interface Builder.  
#  
# NOTE: Do NOT change this file -- Interface Builder maintains it.  
#  
# Put all of your customizations in files called Makefile.preamble  
# and Makefile.postamble (both optional), and Makefile will  
# include them.  
  
#  
  
MAKEFILEDIR = /usr/lib/nib/  
NAME = example1  
INTERFACES = example1.nib  
MFILES = example1_main.m  
LIBS = -lNeXT_s -lsys_s  
INSTALLDIR = $(HOME)/Apps  
ICONSECTIONS = -segcreate __ICON app  
               /usr/lib/nib/default_app_icon.tiff  
  
-include Makefile.preamble  
  
include $(MAKEFILEDIR)Makefile.common  
  
-include Makefile.postamble  
  
-include Makefile.dependencies
```

You shouldn't alter this makefile; Interface Builder maintains it for you. Notice, however, that it lists the name of your application and the source files that are specific to it. It lists the libraries that the linker needs to create the finished application, and it defines the **Apps** directory (within your home directory) as the installation directory for the finished application.

The last four lines let this makefile include as many as four other files:

- Makefile.preamble
- Makefile.common
- Makefile.postamble
- Makefile.dependencies

Makefile.common is always included; the other files are included only if they're present. No error occurs if they're not. **Makefile.common** is the standard NeXT makefile. The ability to include other files lets you add additional rules to this standard makefile.

example1_main.m

This file contains your application's **main()** function:

```
/* Generated by the NeXT Interface Builder. */

#import <stdlib.h>
#import <appkit/Application.h>

void main(int argc, char *argv[])
{
    NXApp = [Application new];
    [NXApp loadNibSection:@"example1.nib" owner:NXApp];
    [NXApp run];
    [NXApp free];
    exit(0);
}
```

There are several points to notice in this short file. First, it includes two header files. The first contains the declaration for the UNIX function **exit()**, and the second includes, directly or indirectly, the declarations and class interface files required by applications using the Application Kit.

The **main()** function starts by creating a new Application object and assigning it to the variable NXApp. This object is sent three messages: The first loads the application's interface file from its location in one of the sections of the executable file, and the second starts the Application object's event loop. At this point the application becomes responsive to the user. When the user chooses the Quit menu command, the event loop terminates and the final message is sent, freeing the application's objects. The last statement calls **exit()**, a standard C library function that terminates the process.

Compiling the Application

Compiling the application is the next step (step 3 in Figure 8-1 above). Choose the Make command in the File menu to start the process.

The Make command sends a message through the Workspace Manager to the Shell application. On behalf of Interface Builder, Shell executes these commands:

```
pushd directory
make debug
popd
```

The first command temporarily changes the current directory to *directory*, the project directory. Next, the **make** command creates a debugging version of the application according to the rules listed in the application's makefile. This version is called **example1.debug**. Finally, **popd** restores the current directory to the one that prevailed before the **pushd** command.

As the **make** command proceeds, you'll see several command lines echoed to the screen as the compiler first compiles and then links the files that make up your application. For more information on the command-line options used by the compiler and linker, see the *Development Tools* manual for more information on the compiler and debugger, or see the UNIX manual entry for **cc**, the GNU C Compiler.

The **make** utility also provides several other useful services besides that of compiling an application. To see what these are, enter:

```
make help
```

The different services, known as targets, are:

```
application (the default)
profile
debug
clean
installsrc SRCROOT=somepath
install [DSTROOT=somepath]
depend
diff SRCROOT=somepath
```

You invoke one of these services by entering the command **make** followed by the name of the appropriate target. Notice that if you don't enter a target name, an optimized version of your application is created by default.

Running the Application

To run your application (step 4 in Figure 8-1), either double-click its icon in a directory window or, in a Terminal or Shell window, switch to the project directory and enter:

```
example1.debug
```

When your application's windows appear, you can verify its operation.

Although limited in scope, this quick application incorporates many of the attributes of a larger program. It responds to mouse and keyboard input and allows simple text editing. In addition, its window can be dragged and resized, and the application can hide itself when the user chooses the Hide command.

Before going on to the next project, you might try altering the interface and then recompiling the application. The recompilation will take little time, since changing the interface only alters the interface file. Before you can run the altered application, the linker simply has to put this new interface file in a segment of the previously compiled executable file.

Project Two: A One-Button Calculator

This project describes how to build a simple calculator using many of the techniques introduced earlier. In addition, it shows how to define a custom object, Calculator, for the application and connect the interface to this object. The calculator's abilities will grow over the course of this project and the next, but its first task will be to convert Celsius temperatures to Fahrenheit. As a temperature converter, the application's calculator window looks like the window shown in Figure 8-8.

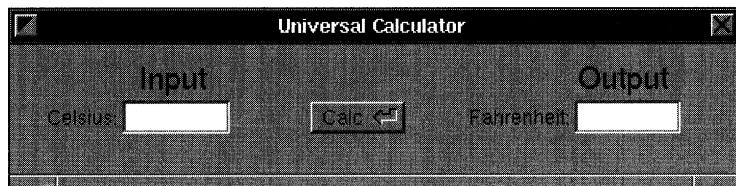


Figure 8-8. The Universal Calculator

The user enters a Celsius temperature in the left text field, then presses Return or clicks the Calc button, and the Fahrenheit equivalent appears in the right field. What happens internally is that when the user signals that the input is complete, the Calculator object takes the input value from the left Form object, performs the calculation, and then sends a message to the right Form object to set its value to the result.

Creating the Interface

As you did in the first project, create a new interface file by choosing the New Application command from Interface Builder's File menu. Notice that a new File window opens and overlaps the File window for **example1** (if this interface file is still open). By allowing multiple interface files to be open at the same time, Interface Builder makes it easy to copy and paste objects from one to the other.

Clicking a File window—or any application window that has an icon in that File window—makes the interface file associated with the File window the current interface file. Commands such as Save or Close File operate on the current interface file. If the interface file is part of a project (and the project is open), that project becomes the current project.

Since the first interface file is finished, close it by making it current and then choosing the Close File command from the File menu. Save the new interface file in a directory called **project2** and a file called **example2**. Remember, you can do this in one step with the Save panel. When the panel opens enter:

```
project2/example2
```

Now, create a project to manage the new application. Click Project in the File menu and, in answer to the question that Interface Builder asks, confirm that you want to create a project file in the **project2** directory.

Next, drag the interface objects shown in Figure 8-8 above from the View palette to the application's standard window. (You'll find it easier to align the different objects if you first turn on the grid feature.)

Edit the text in the Form objects so that they match the titles in Figure 8-8. A single click selects the Form, a double-click selects the field within the Form, and a second double-click lets you edit the text within the field. Edit the two text fields above the Form objects to read "Input" and "Output".

Resize the window so that it resembles the window in Figure 8-8 above. Now open the Window Inspector by dragging to Attributes in the Inspector window's pop-up list. Change the window's title to "Universal Calculator."

With the exception of the Return icon in the Calc button, the Universal Calculator window in your application should look identical to the one in the figure above. To add the icon, open the Icons window (shown in Figure 8-9) by double-clicking the appropriate suitcase icon in the File window.

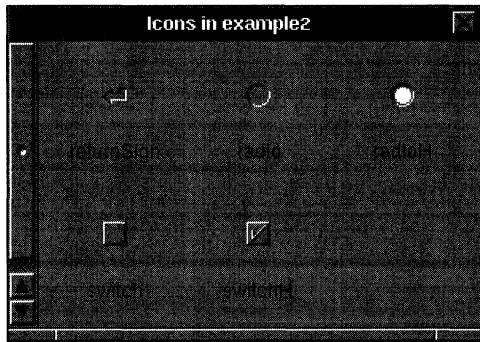


Figure 8-9. The Icons Window

This window displays icons that are used throughout the Application Kit and lets you add new icons by dragging them in from a directory window. Once an icon is displayed in the Icons window, you can drag it onto Button objects in your application. We'll use one of the standard icons in this project; the next project will demonstrate how to create an icon from a file.

Drag a returnSign icon from the Icons window to the Calc button in the Universal Calculator. When the cursor intersects some part of the button, a black rectangle appears around the button to indicate that releasing the mouse button will assign the icon to the button. Release the mouse button and notice that the button resizes to accommodate the title and the icon.

The Button Inspector now lists the name of the button's icon. You can alter the position of the icon in relation to the button's text by using the diamond-shaped Icon Position button at the bottom of the Inspector window. After you press Return to confirm the change, the display of the Calc button is updated to reflect your choice. Try several different placements if you like. If you place the icon above or below the title, the Calc button grows so that both the icon and the title are visible. It doesn't shrink, however, if the extra area is no longer needed. In that case, you have to resize it by hand.

You can also customize the Info panel so that it displays the name of this program and the author.

Defining the Calculator Class

The Calculator object is this application's control center. The Calc button in the interface sends a message to the Calculator object to perform the calculation; in other words, the Calculator object is the target of the Button's action method. The Calculator object must then send messages to the two Form objects to ascertain the input value and set the output value. We'll use the Classes window to design the Calculator class to handle these tasks.

To open the Classes window, double-click the Classes icon in the File window. It appears as in Figure 8-10.

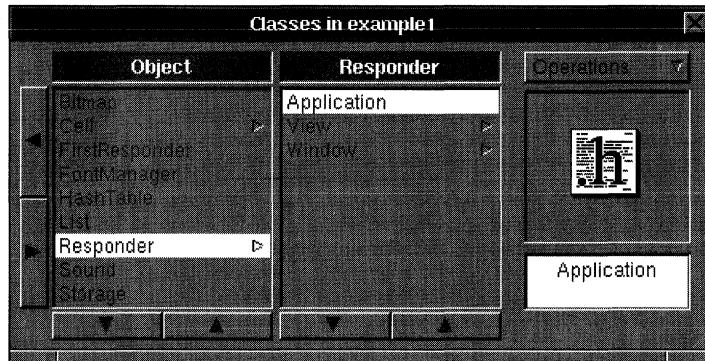


Figure 8-10. The Classes Window

The Classes window displays the class hierarchy of the Application Kit. By pressing the scroll buttons in the browser, you can display different parts of this hierarchy. The class names are displayed in gray, indicating that these classes can't be edited. Notice that the Inspector window that you opened previously now displays the Class Inspector. With this inspector, you can examine (and edit, for classes you build) the outlets and action methods of the class.

Display and select the Application class in the Classes window. Its superclass, Responder, is displayed as the title of one browser column, and the icon for the Application class's interface file appears in the icon well. By double-clicking the icon, you can display the application's interface file in an Edit window. For the classes provided by NeXT these files can't be edited; however, they provide valuable documentation.

With the Application class selected in the Classes window, the Class Inspector displays an Application object's outlet (`delegate`) and the action methods (`hide:` and `terminate:`). Again, these entries are displayed in gray since they aren't editable.

Using the Classes window and the Class Inspector, you can define the class name, superclass, action methods, and outlet instance variables of a custom object. You start defining the new class by selecting where it will go in the class hierarchy. Since a Calculator object has very limited functionality and won't be displayed, we'll make the Calculator class a subclass of Object.

Scroll the browser in the Classes window to the extreme left so that the Object class appears. Click Object, making sure that only this class is selected. The class you define will become a subclass of Object. Now, drag to the Subclass button in the pull-down list. When you release the mouse button, a new class called "Subclass1" appears in the right column. The class name also appears in the text field under the icon well that displays the class icon. Edit this field to read "Calculator" and press Return. Notice that the Classes window now displays the name of the new class in its proper position in the class hierarchy. The name is in black since this class is editable.

Warning: Always check that the intended superclass is selected before you add a subclass. It's easy to inherit from the wrong class.

The next step in defining the Calculator class is to add two outlets corresponding to the Form objects a Calculator object must send messages to. Make sure the Outlet label stands out on the Outlet/Action button in the Class Inspector and then enter “inputForm” in the text field below the button. Click the Add button; the new outlet appears in the Outlets browser. Again, it’s in black, indicating that you can rename or remove this outlet. In the same way add an outlet named “outputForm”.

A Calculator object also needs to respond to an action message from the Calc button in the application. Let’s specify this new action method. Click the Outlet/Action button so that the Action label stands out and then enter “calc:” in the text field. Click Add to add this method name to those displayed in the Actions browser. (Since all action methods take one argument, the **id** of the sender, the method name must end with a colon. If you forget to add a colon, Interface Builder will add one for you.)

This completes the definition of the Calculator class. Interface Builder can now create Objective-C interface and implementation files for the Calculator class. These files will only be templates; we’ll fill them in shortly.

Drag to the Unparse button in the Classes window’s pull-down list. A panel opens asking if you want to create **Calculator.h** and **Calculator.m**. Confirm that you do, and the template files are written into the project directory. Another panel opens asking if you want to add **Calculator.[hm]** (a shorthand for **Calculator.h** and **Calculator.m**) to the project. Again, confirm that you do. The Project Inspector displays the class files within the proper category of the Files display.

In this project, you defined a class and had Interface Builder write template files for the class. The Classes window can also be used to import the class definitions from class files that already exist. Although we won’t try this here, you’d simply drag the icon for the class’s interface file from a directory window to the icon well in the Classes window. Interface Builder will parse the file and add the name of the class in the appropriate position in the class hierarchy. If the new class conflicts with an existing one, Interface Builder gives you the choice of replacing the existing one or canceling the operation. If you want to make this new class part of the project, you must explicitly add it to the Projects window.

Warning: Once you’ve edited a template file, don’t use Unparse again for that class unless you want to overwrite the edited file with a new template file. Interface Builder will warn you before carrying out such an operation.

Now that the Calculator class is defined, you can create an instance of this class—a Calculator object. Verify that the Calculator class is selected in the Classes window and then drag to the Instantiate button in the pull-down list. When you release the mouse button, a new icon appears in the Files window. This icon, titled “CalculatorInstance,” represents your application’s Calculator object. In the next section, you’ll use this icon to make connections between interface objects and the Calculator object.

Connecting the Objects

After gathering the interface objects and creating a Calculator object, you need to interconnect them. To gain an understanding of how objects are interconnected in Interface Builder, let's first look at some of the predefined connections.

We know that the Info command in the application's main menu is connected to the Info panel. To see the connection, display the Info panel by clicking its icon in the File window. Now, click the Info command in your application's main menu to select the top MenuCell. Drag to Connections in the Inspector window's pop-up list to reveal the Connections display for a MenuCell (shown in Figure 8-11).

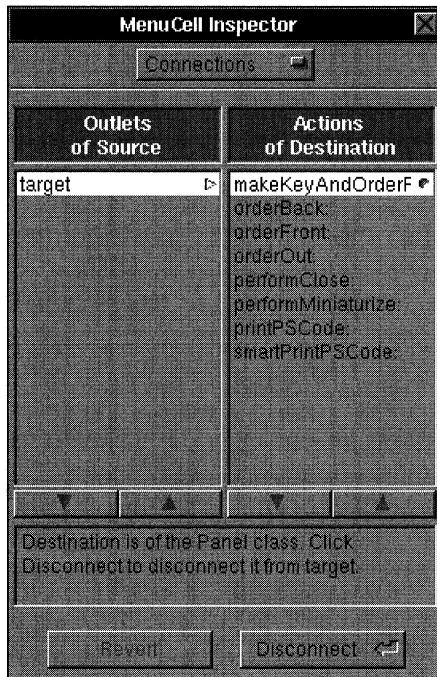


Figure 8-11. The Connections Display

The left column shows the MenuCell's sole outlet, **target**. The right column lists the action messages that the target object, in this case a Panel, responds to. Notice that the entry **makeKeyAndOrderFront:** is highlighted and is marked with a small dimple image. This message, as its name implies, makes the Info panel the key window and displays it above other windows in its tier. The dimple indicates that a connection using this action message has been established previously. The text field near the bottom of the window displays help messages.

To see the connection, click the selection in either column. The connection is displayed in the workspace by black lines drawn between the MenuCell that sends the action message and the Panel that's the target of the message. Figure 8-11 shows this connection.

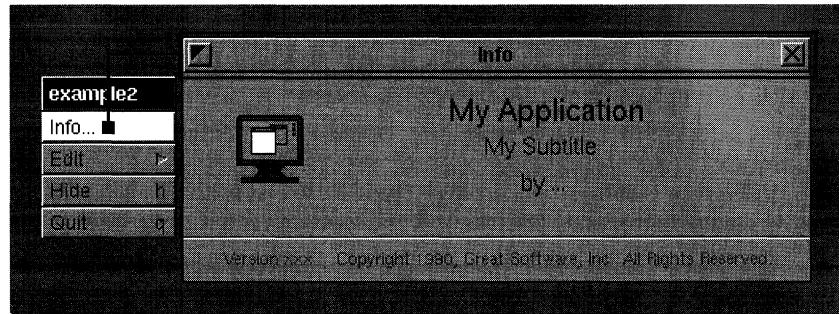


Figure 8-12. Displaying a Connection

Warning: A single click in the Connections displays shows the connection; a double-click removes the connection. Be careful not to remove a connection that you only want to display.

You can also show connections between objects that are part of the visible interface and those that aren't displayed on the screen at run time. For example, to see what the main menu's Hide command is connected to, select the Hide MenuCell. The Connections display verifies that the Hide MenuCell sends a **hide:** message to its target. Click the selection in the Connections display to reveal that the target is the File's Owner. In the same way, you can also determine that the Quit command is connected to the File's Owner.

Now that you've seen how connections are indicated, let's create some in the calculator application. First, let's connect the Celsius Form to the Calc button so that when the user presses Return after entering a Celsius value, the button will act as if it had been clicked. Control-click the Celsius Form and drag the cursor toward the Calc button. You'll notice that a black line trails from the cursor. When the cursor is over the Calc button, release the mouse button. The source and destination of the connection are now identified, and the Form Inspector lists the Form's outlets and the action messages that a Button object responds to. Select the **target** outlet in the first column and the **performClick:** action method in the second column. Finally, click the Connect button to establish the connection. Once a connection is established, you must click the Disconnect button in the Inspector window to remove it.

When the user clicks the Calc button (or it receives a **performClick:** message), the Calculator object should receive a **calc:** message. To identify the source and destination of this connection, Control-click and drag a connecting line from the Calc button to the CalculatorInstance icon in the File window. In the Button Inspector, establish that the Button's target receives a **calc:** action message.

Next, you have to connect the Calculator object's outlets to the appropriate Form objects. Before you do, however, it's important to realize that a Form contains one or more FormCells, and it's possible to connect to the Form or to any of the FormCells within it. In this case, we want to connect to the Form. (See the class specifications for Form and FormCell in *NeXTstep Reference, Volume 2* for more information of these classes.)

To connect the CalculatorInstance to the Celsius Form, Control-drag a line from the CalculatorInstance icon in the File window toward the Celsius Form object. As the cursor first intersects the area of the Form, notice that a gray rectangle appears around the object. As you move the cursor further toward the center of the object, the rectangle changes to black. A gray rectangle indicates that the connection is being made to the Form; a black rectangle indicates that it's being made to the FormCell. Make sure the rectangle is gray and then release the mouse button.

The CustomObject Inspector shows a Calculator object's two outlets, **inputForm** and **outputForm**. Select **inputForm** and click Connect. Notice that a dimple appears adjacent to the **inputForm** listing in the Inspector window. This icon indicates that the connection has been made.

Following the same steps, connect the **outputForm** outlet to the Fahrenheit Form.

If you want to review the target/action connections within your application, select a Control object and then, in the Connections display, click the action message that's marked with a dimple. Connection lines will appear on the screen to identify the object that will receive this message. To review outlet assignments, select the object whose outlets you want to review and click the outlet names in the Connections display. Again, lines will appear on the screen for each connection that's been established.

Save the interface file by choosing the Save command from the File menu. You can now test the interface by choosing the Test Interface command in the File menu. The controls should operate correctly (for example, pressing Return after you enter a number in the Celsius field highlights the Calc button), but of course no calculation takes place. For that, we have to write the Calculator class and then compile the application.

Writing the Calculator Class Definition Files

Interface Builder has given you template files for the Calculator class; now you can add the code that converts from one temperature scale to the other. To open the files, double-click **Calculator.[hm]** in the Project Inspector's Files display. The Edit application opens windows for the class definition files, **Calculator.m** and **Calculator.h**. The Calculator class template files, and the alterations you need to make to them, are described in the next sections.

Calculator.h

The interface to the Calculator class is defined in **Calculator.h**:

```
/* Generated by Interface Builder */

#import <objc/Object.h>

@interface Calculator:Object
{
    id inputForm;
    id outputForm;
}

- setInputForm:anObject;
- setOutputForm:anObject;
- calc:sender;

@end
```

Calculator is a subclass of Object. As you specified in the class editor, a Calculator object has two instance variables that can be used to store the **ids** of the calculator window's input and output fields. Also, as listed in the class editor, a Calculator object has the **calc:** action method.

Two other methods, **setInputForm:** and **setOutputForm:**, appear in the interface file even though we didn't specify them within Interface Builder. Interface Builder automatically provides methods to set the values of each outlet variable. When your application begins running and the interface objects are generated from the archive file, the Application Kit uses these methods to initialize the instance variables to the **ids** of the proper objects.

Calculator.m

Calculator.m contains the implementation of the Calculator class:

```
/* Generated by Interface Builder */

#import "Calculator.h"

@implementation Calculator

- setInputForm:anObject
{
    inputForm = anObject;
    return self;
}
```

```

- setOutputForm:anObject
{
    outputForm = anObject;
    return self;
}

- calc:sender
{
    return self;
}

@end

```

Notice that Interface Builder has implemented the **setInputForm:** and **setOutputForm:** methods for you, but that the **calc:** methods only returns **self**. Next, we'll write the implementation of the **calc:** method.

The **calc:** method must send a message to the object referred to by its **inputForm** variable to retrieve the Celsius value, calculate the Fahrenheit equivalent, and then send a message to the object referred to by its **outputForm** variable to set the value it displays. One implementation of this method looks like this:

```

- calc:sender
{
    float degreesF;
    [inputForm selectTextAt:0];
    degreesF = ((9.0 * [inputForm floatValueAt:0]) / 5.0) + 32.0;
    [outputForm setFloatValue:degreesF at:0];
    return self;
}

```

The first message in this method implementation selects the text in the input field. We select the text so that the user can immediately enter a new value after finishing a previous calculation. Since the field is implemented as a Form object, you need to indicate which entry in the Form object is to be selected. The input Form has only one entry, so the argument to the **selectTextAt:** method is 0. The function of the next two lines should be self-evident. (These lines could be combined into one message, eliminating the **degreesF** variable, but are broken out into two lines for clarity.)

Edit the **Calculator.m** file to include this method implementation. Also, add this directive at the top of the file:

```
#import <appkit/Form.h>
```

This is necessary because the methods referred to in the **calc:** method are defined in **Form.h**. Finally, save the file, and you're ready to compile and test the application.

Compiling and Running the Application

To compile the calculator application, choose Make from the File menu. When the building process has finished, run the application and check its operation.

If your application fails at run time, check over the class definition files to make sure they're correct. If you see either of the following error messages, you've made a simple mistake in connecting objects:

```
error: FormCell Does not recognize selector setFloatValue:at:  
IOT trap (core dumped)
```

```
error: FormCell Does not recognize selector selectTextAt:  
IOT trap (core dumped)
```

Instead of connecting the Calculator's outlets to the Form objects, you've connected them to the FormCells within Form objects. Review "Connecting the Objects" above, correct the connections, and then recompile.

Project Three: Modifying the Calculator

So far, the Universal Calculator can handle any calculation—as long as it's converting degrees Celsius to Fahrenheit. This project describes how to add to the calculator's functionality and, in passing, introduces several features concerning menus and submenus. The final two sections of this project demonstrate how to add icons and sounds to an application.

Let's first create a new project directory for this example and copy the files from **project2** into it. Use the Workspace Manager to make a copy of the **project2** directory. Rename this copy **project3**, and then rename the **example2.nib** and **example2_main.m** files within it **example3.nib** and **example3_main.m**.

Although the directory and files have been renamed, the project file, **IB.proj**, still needs to be updated for these changes. Start Interface Builder (if it isn't already running) and open **example3.nib**. Now, open the Project Inspector and change the application name to "example3" and click OK. Next, choose the Files display. Select and then remove the interface file **example2.nib** and then add the new interface file, **example3.nib**. Save your work.

Now we're ready to begin modifying the copy of the calculator project.

Adding a Submenu

Since the calculator has only one button, extending its functionality beyond temperature conversion means redefining the meaning of the button. (Of course, you could add buttons, but that would be too easy—and wouldn't require a submenu!) The modified calculator application will allow the user to select the type of calculation, either temperature conversion or square root calculation, from a submenu. The titles of the input and output fields will change to reflect the type of calculation selected.

Click the menu button at the top of the Palettes window to display the menu palette. Drag the menu item titled "Submenu" from the Palettes window to the main menu of your application and release the mouse button. The menu item inserts itself within the list of other menu items, and the menu resizes to accommodate the width of the new item. You can reposition a menu item by dragging it vertically within the menu. A submenu containing one menu item appears to the side of the main menu.

MenuCell selection is indicated by highlighting: black text on a white background. Selected MenuCells can be cut, copied, and pasted within a menu or between menus using the standard editing commands.

You can edit the text a MenuCell displays by double-clicking it. Similarly, you can edit the keyboard equivalent for the item by double-clicking the right part of the MenuCell. A square appears indicating that a keyboard equivalent can be added or edited.

Edit the text in the new main menu item so that it reads "Calculations" and press Return. The main menu resizes to accommodate the menu item's text, and the submenu's title changes to match the text. Now add another item to the submenu. Drag the MenuCell titled "Item" from the Palettes window to your application's submenu.

Finally, edit the text of the submenu's two items to read "Temperature" and "Square Root". The finished menus should look like those shown in Figure 8-13.

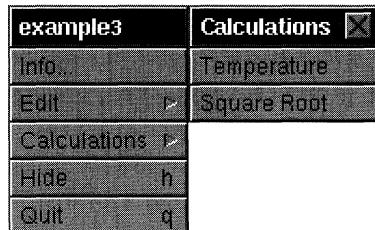


Figure 8-13. The Menu and Submenu

Next, select the Calculator class in the Classes window and then open the Class Inspector to edit the definition of the Calculator class. The revised Calculator object must respond to action messages from the new submenu. Figure 8-14 depicts the Class Inspector and shows the two action methods, `convertToTemp:` and `convertToSqRoot:`, that you need to add to the Calculator class:

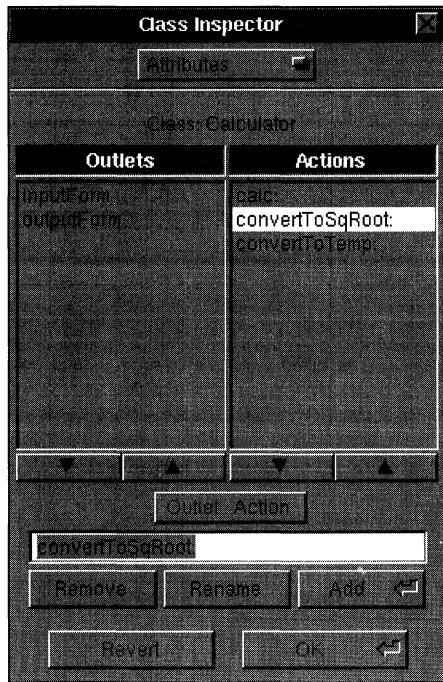


Figure 8-14. Revising the Calculator Class Description

Next, establish the connections from the submenu items to the Calculator object. While holding down Control, drag the cursor from the Temperature submenu item to the CalculatorInstance object in the Files window. Once the connection is established, select the **convertToTemp:** message in the Inspector window and click OK. Likewise, specify that the Square Root submenu item sends a **convertToSqRoot:** message to the Calculator object.

The revised interface is complete; the only changes you have to make concern the Calculator class files. Refer to the Files display in the Projects window. Click the file type for class files. The listing for the Calculator class files appears in the right column. Click Open to display the class files for editing. The next two sections describe the changes you need to make.

Modifying Calculator.h

The new calculator is designed either to convert temperatures or calculate square roots; in other words, the calculator has two states. One way to keep track of the current state of the calculator is to add an instance variable that can have either of two values. We'll add the integer variable **calcType** for this purpose. For convenience, let's also define the constants TEMP and SQROOT to correspond to the two states. These changes add six lines to the **Calculator.h** file. The lines you need to add are shown in bold:

```

/* Generated by the Interface Builder */

#import <objc/Object.h>

#define TEMP 1
#define SQROOT 2

@interface Calculator : Object
{
    id inputForm;
    id outputForm;
    int calcType;
}

+ new;
- setInputForm:anObject;
- setOutputForm:anObject;
- calc:sender;
- convertToTemp:sender;
- convertToSqRoot:sender;

@end

```

Modifying Calculator.m

The implementation file must be modified in three ways. It needs a factory method to initialize the **calcType** instance variable (and thus the calculator's initial state). The **new** method below handles this initialization. When the calculator first appears, it will be configured to perform temperature conversions.

The implementation file also needs a method to respond to **calc:** action messages and perform the proper calculation according to the calculator's present state. The **calc:** method below checks the object's current state before performing the calculation.

Finally, it needs to implement the **convertToTemp:** and **convertToSqRoot:** action methods. These methods set the value of **calcType** and change the titles of the input and output fields.

Make these changes to the **Calculator.m** file. As before, each line you need to add or alter is shown in bold.

```

/* Generated by the NeXT Interface Builder */

#import "Calculator.h"
#import <appkit/Form.h>

extern double sqrt(double);

@implementation Calculator

```

```

+ new
{
    self = [super new];
    calcType = TEMP;
    return self;
}

- setInputForm:anObject
{
    inputForm = anObject;
    return self;
}

- setOutputForm:anObject
{
    outputForm = anObject;
    return self;
}

- calc:sender
{
    [inputForm selectTextAt:0];
    if (calcType == TEMP) {
        float degreesF;
        degreesF = ((9.0 * [inputForm floatValue])/5.0) + 32.0;
        [outputForm setFloatValue:degreesF at:0];
    } else if (calcType == SQROOT) {
        double sqRoot;
        sqRoot = sqrt((double)[inputForm floatValueAt:0]);
        [outputForm setFloatValue:(float)sqRoot at:0];
    }
    return self;
}

- convertToTemp:sender
{
    /* label input and output fields */
    calcType = TEMP;
    [inputForm setTitle:"Celsius:" at:0];
    [outputForm setTitle:"Fahrenheit:" at:0];
    [outputForm setStringValue:@"" at:0];
    [inputForm display];
    [outputForm display];
    [inputForm selectTextAt:0];
    return self;
}

```

```

- convertToSqRoot:sender
{
    /* label input and output fields */
    calcType = SQROOT;
    [inputForm setTitle:"x:" at:0];
    [outputForm setTitle:"sqrt(x):" at:0];
    [outputForm setStringValue:@"" at:0];
    [inputForm display];
    [outputForm display];
    [inputForm selectTextAt:0];
    return self;
}

@end

```

After you edit and save these files, compile the application. Watch for error messages from the compiler. In most cases, they will signal typographical errors in the source code. Make the necessary corrections and recompile the application. Finally, run the application and test its new features.

Note: If the application fails at run time, the problem is probably caused by an inconsistency between the method and instance variable names you declared in the class editor and those in the Calculator class definition files. Restart Interface Builder and check the method and variable names in the Class Editor panel against those in **Calculator.h** and **Calculator.m**.

Adding an Icon

With the Icons window, you can access existing system icons, as illustrated in the previous project, or you can create icons from data in either TIFF (Tag Image File Format) or Encapsulated PostScript (EPS) file format. Once you import the image data, it becomes an icon that you can assign to Button objects in your application. Figure 8-15 shows some examples of buttons that display icons.



Figure 8-15. Icons and Buttons

To see how this works, double-click the Icons suitcase in the Files window. The Icons window appears and displays a variety of icons used in the Application Kit. The titles under the icons are displayed in gray to indicate that these icons can't be deleted nor can their names be edited. However, you can copy and paste any icon that appears in this window.

Let's add an icon to this window. Using one of the Workspace Manager's directory windows, switch to

/NextLibrary/Documentation/NextDev/Examples/IBTutorial/Images

You'll notice that this directory contains the TIFF file **willy.tiff**. Drag the file icon from the directory window to Interface Builder's Icon window. When you release the mouse button, Interface Builder reads the image data from the file and constructs an icon. If the icon is no larger than 48-by-48 pixels, the Icons window shows the actual image. Otherwise, (as in this case) it displays a generic icon. Only the image data—not the file **willy.tiff**—is added to the project.

The Inspector window changes to show the Icon Inspector. This display has two uses: It gives you the dimensions of the image in pixels, and it lets you see the actual icon image even for icons larger than 48-by-48 pixels. Figure 8-16 shows a detail of the Icon Inspector.

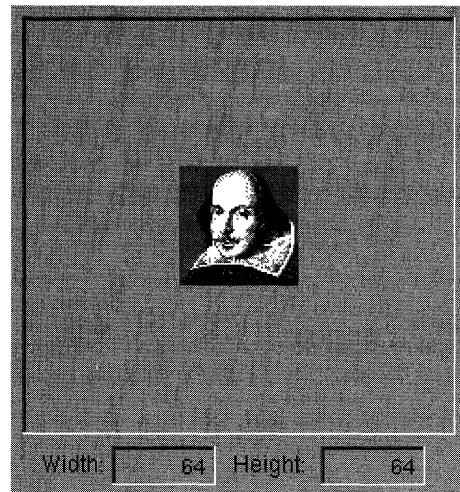


Figure 8-16. The Icon Inspector

To place the icon on a button in your application, simply drag the icon from the Icons window to a Button object in your application's window. (The cursor must be over the button when you release the mouse button; otherwise, the icon isn't transferred.) For example, open the Info panel and select the generic application icon in it. If you check the Attributes display of the Inspector window, you'll see that this icon (named **defaultappicon**) is mounted on a button object. Now, drag the Shakespeare icon from the Icons window to the Info panel and deposit it on this button. The Shakespeare icon replaces the application icon.

Adding Sound

To manipulate the sounds in your application, Interface Builder provides two tools, the Sounds window and the Sounds Inspector. The Sounds window is the repository for your application's sound resources. By dragging a sound icon from the Sounds window onto a Button object in your application, you can associate a sound with that object. The Sound Inspector lets you play prerecorded sounds from sound files on disk and lets you record your own sounds. It also gives you a graphic display of the sound and allows you basic editing capability.

Open the Sounds window by double-clicking the Sounds icon in the Files window. Each of the icons in the Sounds window represents a sound. The gray titles indicate that these sounds can't be edited since they are system sounds. You select a sound by clicking its icon. A selected sound can be copied, pasted, and (except for system sounds) deleted. In fact, it's common to create a new sound for editing by copying an existing sound.

Make a copy of the Basso sound in the Sound window. The new sound icon is labeled "Sound." Now, open the Sound Inspector by double-clicking the new sound's icon.

The Sound Inspector shows a graphical representation of the selected sound's waveform. The graph plots the sound's amplitude versus time. You can play the entire sound file by clicking the Play button, or you can select and play only a portion of the displayed sound. For a demonstration, drag horizontally across a portion of the graph and click Play. Notice that the sound meter below the waveform shows the instantaneous and peak volumes for the sound that's played.

If your machine has a microphone, you can replace the selection in the Sound Inspector with sound you record. Make sure the microphone is on and then click the Record button to start recording. When you're through recording, click Stop to end the recording session and display the wave form. Clicking Pause halts the recording until the next time Pause is clicked.

You can add sounds to the Sounds window by dragging the sounds file icon from a directory window to the Sounds window. As with icons, only the data from the sound file becomes part of the project; the file itself isn't copied into the project directory.

Using a directory window, switch to the

`/NextLibrary/Documentation/NextDev/Examples/IBTutorial/Sounds`

directory. Within this directory, there are three sound files: **drum1.snd**, **drum2.snd**, and **drum3.snd**. Drag **drum1.snd** into the Sounds window. The graph in the Sounds Inspector shows the sound's waveform.

Now, let's create a sound for the Calc button in the Calculator application. Select a portion of the drum sound. For example, you might find that the decay portion of one of the louder drum beats, as shown in Figure 8-17, makes a satisfying button click sound.

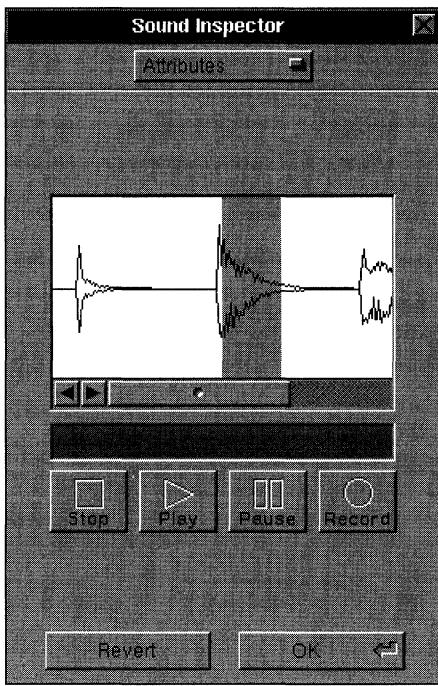


Figure 8-17. The Sounds Inspector

Once you've found a portion of the waveform that you want for the Calc button, select and then delete the portions that precede and follow it. Click OK to save the modified sound.

Next, let's associate the sound with the Calc button. Drag the sound icon from the Sounds window to the Calc button and release the mouse button. The sound is played and the button becomes selected to confirm that the sound has been assigned to the button. If you look at the button's attributes in the Button Inspector, you'll see that **drum1** is listed. By deleting this name, you can remove the association of the sound with the button. You can check the operation of the button by putting Interface Builder in test mode and then clicking Calc.

This ends the Universal Calculator project. Save the project and then compile and run the application to test its operation. You might try adding other features to the calculator to test your understanding of the concepts introduced so far.

Project Four: A Text Editor Using Two Interface Files

Most larger applications benefit from storing different parts of their interface in separate interface files. The primary elements of the interface—the main menu and perhaps a window or two—are described in one interface file, and the other parts of the interface are described in one or more auxiliary interface files. When the application starts, its primary interface objects are created immediately. Objects specified in its auxiliary interface files are created only on demand, as when a user requests a help window, for example.

This program design is a consequence of the way interface files are accessed by an application. As you've seen in the earlier projects, all objects described in an interface file are created at the same time:

```
[NXApp loadNibSection:@"example1.nib" owner:NXApp];
```

There's no way to load a subset of an interface file's objects. However, the same functionality can be gained by using multiple interface files.

Using multiple interface files can improve your application's perceived performance. If at startup time, an application creates only those objects a user will need immediately, the time it takes to start the application can be reduced. Of course, as users attempt to access other parts of the application, they will experience small delays as new objects are created from the auxiliary interface files. However, these delays are minimal and are incurred only when a user requests a specific part of the interface, rather than being imposed indiscriminately on all users when the application starts.

An equally important reason to have more than one interface file is to let an application replicate a piece of its interface an indeterminate number of times. The document windows in Edit are a good example. Since it can't be predicted how many document windows a user might need, the application must provide a way to create an unlimited number of them. By putting the document window interface in a separate interface file, each time a user requests another window, a new set of objects can be created from the file.

To give you a better understanding of how to design an application using multiple interface files, this project introduces a simple, multiple-document, text editor that's implemented using two interface files.

The Text Editor's Design

The text editor has a simple design: Through the application's Window menu, a user can open any number of document windows. Text entered in a document window can be cut, copied, and pasted using the Edit menu. With one document window open, the application presents this interface:

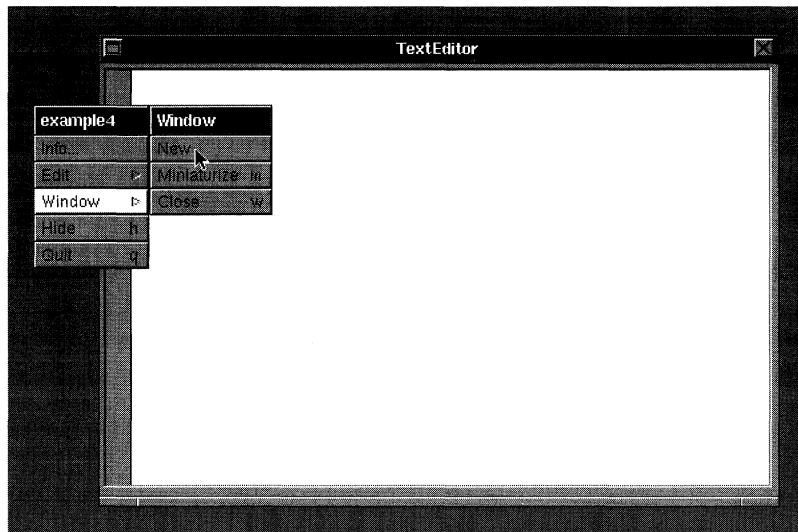


Figure 8-18. The Text Editor

The interface you see in Figure 8-18 is actually created using two interface files. The primary interface file contains the specification for the application's main menu and its submenus. The specification for a document window and its scrolling text area is contained in a separate interface file. The two interfaces are linked by two custom objects (one of the Distributor class and one of the TextEditor class), as shown in Figure 8-19.

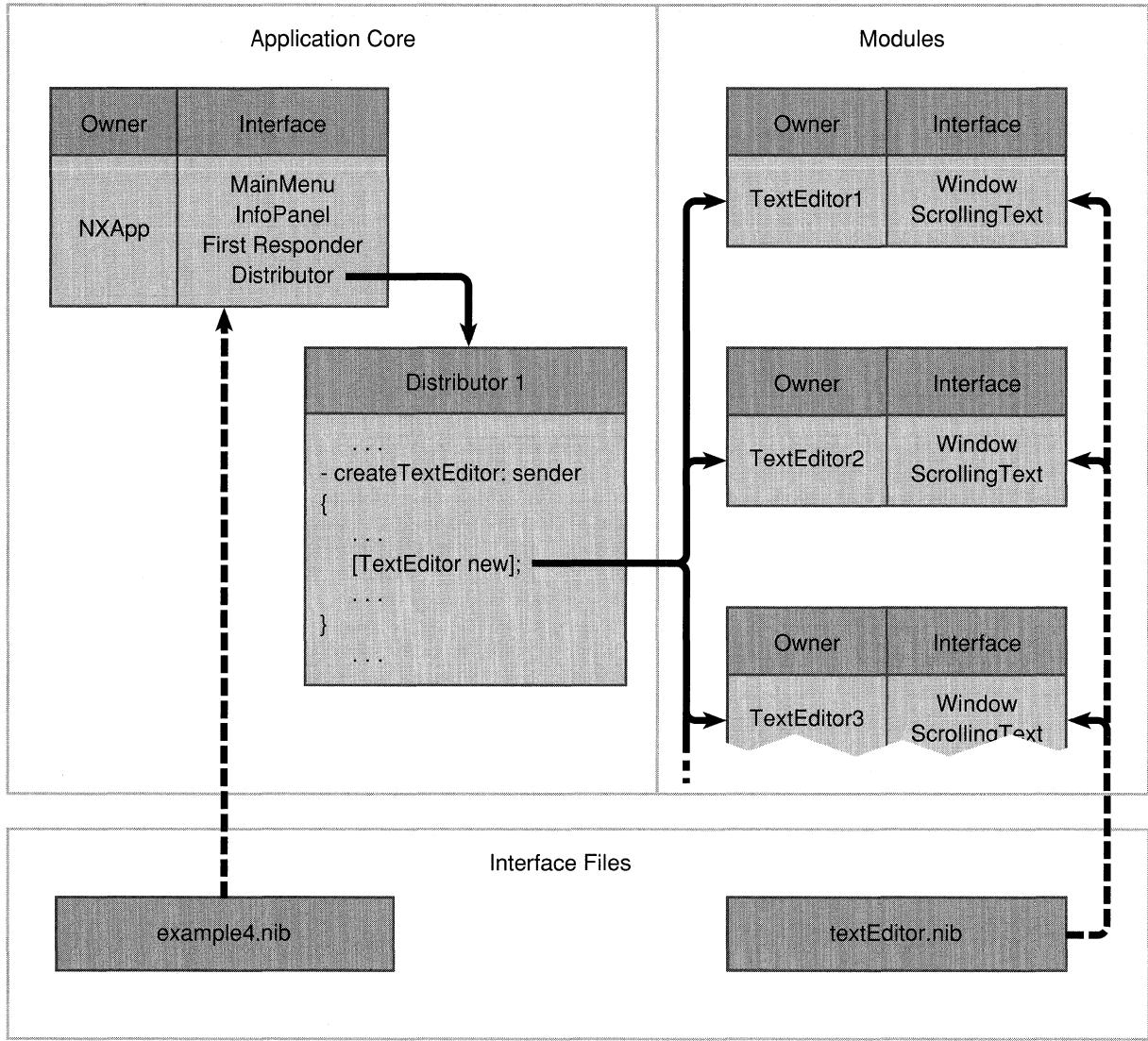


Figure 8-19. The Application's Design

When the application starts, the primary interface objects are created from the specification in **example4.nib** and connected to their owner, **NXApp**. An object of the **Distributor** class, a class that you'll implement, is also created. So far, only the main menu appears on the screen, although the objects that make up the submenus have also been created. When a user clicks the New command in the Window menu, a **createTextEditor:** action message is sent to the **Distributor** object. As you can see in the figure, this object in turn creates a new object of the **TextEdit** class, another class you'll define in the process of building this application.

The **new** method in the `TextEditor` class contains these lines:

```
+ new
{
    self = [super new];
    [NXApp loadNibSection:@"textEditor.nib" owner:self];
    return self;
}
```

Each `TextEditor` object, as it's created, is made the owner of a set of objects specified in the **textEditor.nib** interface file. Thus, each time the user clicks the New command, a new `TextEditor` object along with a new window and scrolling text area are created.

This is the general pattern for using multiple interface files in an application. The application's core has its own interface. Similarly, each module minimally consists of a custom object and its interface. When a new module is required, an object within the application's core creates the module's custom object, which loads its own interface.

Note: It would be possible for an object within the application's core to load the module's interface objects directly, eliminating the need for the module's custom object. For example, the `Distributor` object could load, and make itself the owner of, each module interface. However, it's generally better programming practice to have a module contain both an interface and its underlying logic. In this way, a module can be truly independent of the application's core objects, storing any pertinent state information in its owner. If the application needs information about a module's state, it can query the module's owner.

Creating the Application's Interface

Before you start building this application, close any other interface files you may still have open. Next, choose the New Application command from the File menu. By default, a main menu and a window appear. As you can see in Figure 8-19 above, however, the application's primary interface file doesn't include a window—windows are provided by the auxiliary interface file. Remove the window by selecting it (either by clicking it or by selecting its icon in the File window) and then choosing the Cut command. Since most applications have at least one standard window, a panel opens asking if you really want to remove this window. Confirm that you do.

Now, let's add some commands to the main menu. Click the menu button in the Palettes window and drag a Window menu item to your application's main menu. Position this item immediately above Edit. The Window menu that opens displays more commands than you'll need in this project. Cut all but the New, Miniaturize, and Close commands from the menu.

This completes the visible part of the interface for the application's core. Save the interface file in a directory called **project4** and a file called **example4**. (Enter **project4/example4** in the Save panel to do both of these steps at one time.) Make a project for this application by choosing the Project command in the File menu. Next, we'll define the invisible part of the interface, the `Distributor` class.

Defining the Distributor Class

The Distributor class, a subclass of Object, defines an object whose primary function is to create new document windows whenever a user chooses the New command from the application's Window menu. As it creates each new document window module, it temporarily stores the identity of the module's owner. In a more robust application, the Distributor class would keep track of each module's owner so that it could later "distribute" messages from the application's core objects to any one of the modules.

To create the Distributor class, open the Classes window and scroll to the left to reveal the Object class. Click the Object class so it's the only class that's selected in the browser. Now, create a subclass of Object by dragging to Subclass in the pull-down list. When you release the mouse button, a new class is inserted in the class hierarchy. Change the name of this class to "Distributor".

The next step is to define the Distributor class's single action method. Open the Class Inspector. (Since the Project Inspector is already open, simply drag to Attributes in its pop-up list.) Now, define a Distributor object's single action method by clicking the Outlet/Action button in the Class Inspector so that Action stands out and then entering **createTextEditor:** in the text field. Click Add.

To create template source code files for the Distributor class, drag to Unparse in the Classes window pull-down list. Two panels open in succession: The first ask you to confirm that you want to write these class files, and the second asks whether these class files should be added to the project. Click OK in both panels. If you look at the Files display in the Project Inspector, you'll notice that the files **Distributor.[hm]** are listed under the class files.

Editing the Class Files

Use Edit to modify the Distributor class definition files. Remember that you can double-click the file name in the Files display of the Project Inspector to open Edit windows for both **Distributor.h** and **Distributor.m**. Add the lines that appear in bold in the listings below. An explanation of these additions follows the listings.

Make these changes to the class interface file, **Distributor.h**:

```
/* Generated by Interface Builder */

#import <objc/Object.h>

@interface Distributor:Object
{
    id newTextEditor;
}

- createTextEditor:sender;
@end
```

Also, make these changes to the class implementation file, **Distributor.m**:

```
/* Generated by Interface Builder */
#import "Distributor.h"
#import "TextEditor.h"

@implementation Distributor

- createTextEditor:sender
{
    newTextEditor = [TextEditor new];
    [newTextEditor show:self];
    return self;
}

@end
```

Each time a Distributor object receives a **createTextEditor:** message, it creates a new TextEditor object (the owner of the document window module) and stores the object's **id** in its **newTextEditor** variable. Next, it sends a **show:** message to the new TextEditor object. As you'll see when you define the TextEditor class, this message brings the module's document window to the front of its tier on the screen and makes it the key window.

Notice that the **newTextEditor** variable isn't an outlet. Outlets are associated with interface files and are defined in Interface Builder. At run time, an outlet is initialized to the **id** of some object in the interface file or to the owner of the interface file. A Distributor object's **newTextEditor** variable, on the other hand, is not associated with an interface file. It's used to cache the **id** of a TextEditor object temporarily. As suggested earlier, in a more complex application, the Distributor object might keep track of each TextEditor object it creates so that it can send messages to any one of them. For example, it might use an object of the List class (one of the Application Kit classes) or the HashTable class (one of the common classes defined in **libobjc_s.a**) to retain the **ids** of each of the TextEditor objects it creates.

After you've made these changes to the class files, save them and close the Edit windows.

Connecting the Objects

Now that the Distributor class is defined, it's time to create a Distributor object and connect it to the New command. With the Distributor class selected in the Classes window, drag to Instantiate in the pull-down list. When you release the mouse button, notice that an icon for the custom object appears in the File window. This icon is titled **DistributorInstance**.

Control-drag a connection from the New command in your application's Window menu to the **DistributorInstance** icon. The Inspector window shows the Connections display for the MenuCell inspector. Make sure the **target** outlet and the **createTextEditor:** action are selected and click Connect. Now, whenever the user chooses New, a **createTextEditor:** action message will be sent to the Distributor object.

This completes the first half of this project; next you'll create the application's document module. Before going on, save your work and, if you like, clean up the workspace by closing the **example4.nib** interface file. (Choose Close File from the File menu.) You can also close the Inspector window.

Creating the Module's Interface

A module consists of a window, a scrolling text area, and a custom object that owns the interface. The custom object will be of the **TextEditor** class, a subclass of **Object** that you'll define shortly.

Choose the New Module command from the File menu. A panel opens letting you set the class of the interface file's owner. Enter "TextEditor" in the Name field. You can also specify whether the **TextEditor** class is an immediate subclass of **Object** or **Application**. Leave the Subclass of Object button selected and click OK to register these choices.

Note: In most cases, a module's owner should be an immediate subclass of **Object**. See "New Module" in the reference section that follows for an explanation.

The File window shows that this module initially consists of an interface file (**Untitled1.nib**) that contains an owner object, a first responder object, and various resources. Drag a window from the Palettes window into the workspace and open the Window Inspector. Change the title of the window to "TextEditor." Now, drag a **ScrollView** object into the window and resize it so that it covers most of the window's area. Save the interface you've created so far in a file named **textEditor.nib**. Finally, add this interface file to the list of interface (.nib) files in the Files display of the Project Inspector.

The visible portion of the module's interface is complete; the next job is to define the owner object.

Defining the **TextEditor** Class

When you started building this module, you specified that the owner object was of the **TextEditor** class, a subclass of **Object**. Interface Builder has already added this new class to the hierarchy in the Classes window. Open the Classes window and scroll to the **TextEditor** entry.

The next step is to define the outlets and actions of this new class. Make sure the **TextEditor** class is selected in the Classes window and then open the Class Inspector. Following the same general steps you took with the **Distributor** class, give the **TextEditor** class a **myWindow** outlet and a **show:** action method. Create class definition files for the **TextEditor** (that is, drag to the Unparse button in the Classes window) and add these files to the project.

Editing the Class Files

As before, edit the class interface file **TextEditor.h** by adding the lines that appear in bold:

```
/* Generated by Interface Builder */

#import <objc/Object.h>

@interface TextEditor:Object
{
    id myWindow;
}

+ new;
- setMyWindow:anObject;
- show:sender;
@end
```

Also, make these changes to the class implementation file **TextEditor.m**:

```
/* Generated by Interface Builder */

#import "TextEditor.h"
#import <appkit/Application.h>
#import <appkit/Window.h>

@implementation TextEditor

+ new
{
    self = [super new];
    [NXApp loadNibSection:@"textEditor.nib" owner:self];
    return self;
}

- setMyWindow:anObject
{
    myWindow = anObject;
    return self;
}

- show:sender
{
    [myWindow makeKeyAndOrderFront:self];
    return self;
}

@end
```

The **new** method creates a TextEditor object and makes it the owner of the module's interface. The **show:** method sends a **makeKeyAndOrderFront:** message to the window in the interface through the TextEditor's **myWindow** outlet. Since **TextEditor.m** sends messages to objects of both the Application class (through NXApp) and the Window class, the header files for these classes are included with the **#import** directives at the top of the file.

Connecting the Objects

A TextEditor object is connected to the other objects in the application in two ways: through the **show:** action message that it will receive from the Distributor object and through the **myWindow** outlet that will be initialized to the **id** of the window in the module's interface. You've already written the code in **Distributor.m** that sends the **show:** message to a TextEditor object; that connection is complete.

Connect the owner object's **myWindow** outlet by Control-dragging a connection from the owner's icon in the File window to the title bar of the document window. Select **myWindow** in the Inspector window's Connections display and click OK.

Now that all the pieces are in place and the connections are established, it's time to compile and test the application. Before you do, you may want to clean up workspace by closing any of Interface Builder's windows you no longer need.

Compiling and Running

Choose Make from the File menu. If an interface file in this project needs to be saved, Interface Builder asks if you want to save it before proceeding.

When the compilation process is finished, run the application and test its operation. Each time you choose the New command, a new window opens directly on top of the old one. If you click in the scrolling text area, a blinking vertical bar appears marking the insertion point. Check other features such as text entry and editing, pasting text between windows of this application (and between this and other applications), and window resizing.

Although this completes the text editor project, this application provides a good basis for exploring other features of the Application Kit. Perhaps the easiest improvement would be to add a Font command to the main menu. (Remember, however, that the Text object in the ScrollView currently supports only one font. Use the ScrollView Inspector to change this setting.) You could also, for example, implement the Window commands that you previously deleted, such as Open and Save. Or you might try making a simple graphics editor that supports multiple documents. (Hint: Use a CustomView object in the document window. In the class implementation of the CustomView, implement the **mouseDown:** method.)

Interface Builder Reference

The following sections discuss all of Interface Builder's commands and windows. The menus are presented first, with each command being described in the order of its appearance. Interface Builder's windows and panels are discussed next. Since several of these windows support multiple displays (for example, the Palettes window has three separate displays: View, Windows, and Menus), each display is discussed in its own section.

The Main Menu

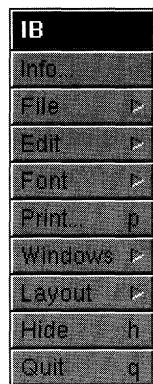


Figure 8-20. The Main Menu

Info

The Info command opens a panel that displays information about Interface Builder. Besides identifying the application, the panel shows the names of its authors, a copyright notice, and a version number. Clicking an author's name displays a picture of that author.

File

The File command opens a menu containing commands that operate on files. These commands let you create a new interface file or open an existing one, save your work to the currently open file or to another file, close an interface file or a window, create a new project file, and add interface files to the currently open project file. The File menu also contains the Test Interface command. This command puts Interface Builder in test mode so you can check the operation of your application's interface.

Edit

The Edit command opens a menu containing the standard editing commands Cut, Copy, Paste, Delete, and Select All. These commands can be used to edit text, objects, and icons within Interface Builder.

Font

The Font command opens a menu that contains the standard Font menu commands. With this menu, you can alter the font of text within the windows of your application. You can't set the font of text that can't be selected, such as the text that appears in a window's title bar.

Print

The Print command opens a standard Print panel that lets you print a window and its contents. The window that's printed is the main window, either one of the windows in the application you're building or one of Interface Builder's windows. This command is useful for documenting the application's interface or the stages of the development process.

Windows

The Windows command opens a menu that gives you access to the Palettes, Inspector, and Page Layout windows. In addition, this menu contains the Close Window command, which closes the key window.

Layout

The Layout command opens a menu that gives you access to many of Interface Builder's drawing commands. These commands let you adjust the order of overlapping objects, resize an object to fit its contents or to match the size of another object, or group objects visually by placing a box around them. The Layout menu also lets you control Interface Builder's grid feature, which helps you align objects within a window.

Hide

The Hide command removes Interface Builder's windows, and all windows of the application you're developing, from the screen. Double-clicking Interface Builder's application icon returns these windows to the screen.

Quit

The Quit command exits Interface Builder. If there are any interface files that haven't been saved, a panel opens giving you the opportunity to save them before quitting Interface Builder.

The File Menu

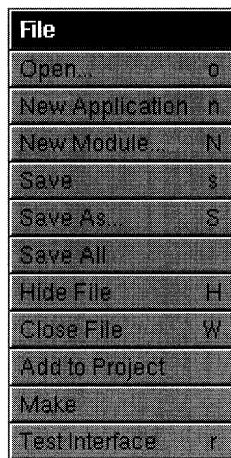


Figure 8-21. The File Menu

Open

The Open command displays an Open panel that lets you open an existing interface or project file. The only files displayed in this panel are interface and project files.

When you open an interface file, a File window appears displaying icons that represent the contents of the interface file. The windows that were visible when the interface file was last saved again appear in the workspace. If the interface file is part of a project, the project file is opened automatically.

The newly opened file becomes the current interface file. File commands such as Close or Hide File will act solely on it. Other open interface files still display their windows, but you must click one of these windows to make its interface file current before doing additional work in the file.

If you use Open to open a project file containing a main interface file, the main interface file is opened too.

New Application

The New Application command starts a new application interface file and makes it the current interface file. This command provides you with the basic interface objects of a new application:

- A main menu
- A standard window
- An Info panel

The main menu contains the Info, Edit, Hide, and Quit commands. These commands are connected so that they send the appropriate action messages to their targets, as you can verify by examining the connections using the Inspector window. (See “The Inspector Window” below for more information.) The standard window is empty, displaying only a light gray background. The Info panel isn’t displayed but can be made visible by double-clicking its icon in the File window. The Info panel contains three text fields and a button that’s used as a frame for an icon representing the application or its developer.

The File window also displays the name “First Responder,” which represents the object in a window that has first responder status. See the “First Responder” section later in this chapter.

When you choose the New Application command for the first time, the File window’s title displays the path of the file named **Untitled1**, the default name for the new application’s interface file. You can change the path and name when you save the file, by using either the Save or the Save As command.

A new application’s interface file isn’t automatically part of a project; it must be explicitly added by choosing the Project Inspector.

New Module

The New Module command starts a new interface file for a module of an application and makes it the current interface file. This command is the starting point for building an application module, such as a help system or a reusable document window. Unlike the New Application command, which provides a complete (but minimal) set of interface objects for a standard application, the New Module command gives you an interface file that has references only to an owner and a first responder object.

When you choose the New Module command, a panel opens letting you set the name and class of the owner of the interface file. (See Figure 8-22.)

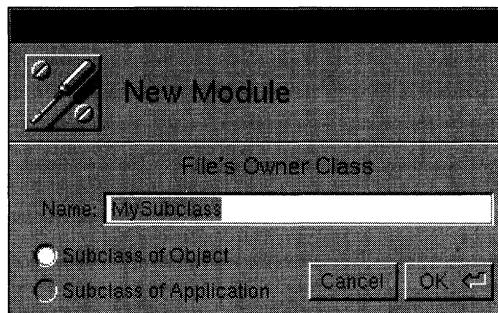


Figure 8-22. The New Module Panel

You can change the name of the owner's class, which by default is MySubclass, and choose whether the owner's class will be a subclass of Object or Application. If necessary, your choices can be revised later using the Class Inspector.

It's generally better to make the owner's class a subclass of Object rather than Application since this makes the module more easily reusable in other applications. A brief explanation will make this clear. The owner of an interface file typically has a number of outlets and action methods that are provided for the objects within the interface file. For example, the owner of the interface file for a help system might have outlets that point to each of the windows in the help system. To reuse this help system in a different application requires that the new application instantiate the owner object which in turn loads the help system's interface file. This is no problem if the owner is an object that's a subclass of Object—there's no restriction on how many objects of this type an application can have. However, there is a problem if the owner is an Application object since an application can have only one Application object. To reuse the help system in this case demands that you modify the application's Application object to include the outlets and action methods that the help system interface files requires. For an example of a reusable module, see the fourth project in the preceding tutorial section.

After you click OK in the New Module panel, the File window opens, showing you the objects and resources in the interface file. The File window's title displays the path of the file named **Untitled1**, the default name for the new module's interface file. You can change the path and name when you save the file, by using either the Save or the Save As command.

A new module's interface file isn't automatically part of a project; it must be explicitly added by using the Project Inspector.

Save

The Save command saves the data for the current interface file. If the interface data has been saved previously, the Save command simply writes the data to the file and directory location displayed in the File window's title. If the data hasn't been saved previously, a Save panel opens letting you specify a file name and directory location for the interface data.

Save As

The Save As command saves the data for the current interface file under a different file name or in a different directory (or both), and lets you continue working in this new file. The previous file is closed and is not affected by the subsequent work you do in the new file.

Save All

The Save All command saves all open interface files. For each interface file that hasn't been saved previously, a Save panel opens letting you specify a file name and a directory location for the file.

Hide File

The Hide File command hides all windows associated with the current interface file. The interface file's File window is hidden along with any menus, standard windows, or panels the interface may contain. This command is handy for cleaning up the workspace when you're editing multiple interface files.

The hidden interface windows are represented by a single miniwindow that appears at the bottom of the workspace. The miniwindow's title bar displays the name of the interface file. Double-clicking the miniwindow causes the hidden windows to reappear and the file to become the current interface file.

Close File

The Close File command closes the current interface file. If changes have been made to the file since it was last saved, a panel opens giving you the opportunity to save the file before closing it.

Project

The Project command opens the Project Inspector. If there's no project file in the current directory, the Project Inspector asks if you want to create one. Once created, the project file is used to keep track of your application's components. See "Project Inspector" later in this chapter for more information.

Make

The Make command starts the process of building an executable version of your application. You must have these components before the Make command is enabled:

- A interface file
- A project file
- A main file
- A makefile

You create the interface file using the New Application or New Module command in the File menu. Interface Builder provides you with the other files through the Project Inspector, see “The Project Inspector” later in this chapter for more information.

Assuming you have these components, choosing the Make command sends a message (through the Workspace Manager) to the Shell application. A Shell window comes to the front and executes these commands:

Command	Effect
<code>pushd <i>directory</i></code>	Temporarily changes the Shell window’s current directory to <i>directory</i> , the project directory.
<code>make debug</code>	Creates an executable version of your application with a file name of the format <i>application.debug</i> . This version contains the symbols required by the debugger GDB for debugging your application.
<code>popd</code>	Reverses the effect of the previous pushd command and restores the previous directory as the current directory. (See the UNIX manual entry for csh for more information on pushd and popd .)

After your application has been built, you can run it either from the Shell window or from the Workspace Manager. If you use Shell, remember to switch to the application’s project directory before attempting to run your application.

During the application development process, the Make command provides a convenient way to create a debugging version of your application. However, the debugging version will take more space than the final version, so once you’re confident the application runs correctly, create a version without debugging symbols. To do this, in a Shell or Terminal window switch to the project directory and enter:

```
make [application name]
```

Test Interface

The Test Interface command puts Interface Builder in test mode. When you choose this command, Interface Builder's support windows disappear, leaving only those windows that belong to your application. You can then test the operation of the Application Kit objects within your application. For example, normally in Interface Builder, clicking a radio button in one of your application's windows selects the object for further editing. You can reposition it, increase or decrease the number of ButtonCells the Matrix contains, or alter the spacing between the cells. In test mode, such a click selects a button within the radio group just as it would if the application were running by itself. If you have specified an action message and target for the button, the message is dispatched to its target. If the target is a Kit object within the interface, the object responds to the message. Messages to objects outside the interface have no effect in test mode.

When your application is operating in test mode, Interface Builder's application icon changes to display a large switch (shown in Figure 8-23).

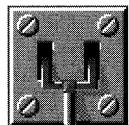


Figure 8-23. The Test Mode Icon

To exit test mode, either choose the Quit command in your application's main menu (if present) or double-click this switch icon.

The Edit Menu

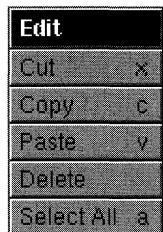


Figure 8-24. The Edit Menu

Cut

The Cut command deletes the selection and puts it on the pasteboard. Besides acting on selected text, this command also works on objects in your application's windows and on icons in the File, Sounds, and Icons windows.

Several objects are protected to different degrees from being cut. Each interface file must contain references to an owner and a first responder object, so Interface Builder doesn't allow you to cut the icons for these objects. The other icons in the File window may be cut. However, since cutting a window deletes it as well as the objects it contains, Interface Builder asks for confirmation before carrying out the Cut command when a window is selected.

Cutting an object severs any connections between it and other objects in the application. Pasting the object back into the interface doesn't restore these connections.

Copy

The Copy command puts a copy of the current selection on the pasteboard. Besides acting on selected text, this command also works on objects in your application's windows and on icons in the File, Sounds, and Icons windows.

Since an interface file can only have one reference to an owner object and to a first responder object, and it can have only one set of resources, Interface Builder doesn't let you copy the icons for these objects. You can copy any of the other icons in the File window. In fact, the quickest way to duplicate a window and its contents is to copy and paste its icon in the File window. You can also use this command to copy icons and sounds in the Icons and Sounds windows.

Copying an object in your application also copies any sound or icon associated with it. However, the Copy command doesn't copy the connection between the selected object and another object.

Paste

The Paste command inserts a copy of the contents of the pasteboard. Pasting text replaces the current selection with the text from the pasteboard. Pasting an object or icon simply adds a copy of the object or icon to the key window.

Interface Builder only lets you paste the contents of the pasteboard to an appropriate destination. For example, you can copy a sound from the Sounds window's system sounds display and paste it back into the Sounds window to create another sound, but you can't paste the sound into the File window.

Delete

The Delete command deletes the selection without putting it on the pasteboard. The same restrictions apply for this command as for the Cut command; see the section on the Cut command above for more information.

Select All

The Select All command selects all of the text or objects in the key window.

If your application's key window contains an object that has a text selection, the Select All command causes the selection to grow to encompass the entire contents of that object. If no object displays a text selection, the Select All command causes all of the objects within the key window to become selected. Once the objects are selected, they can be moved as a group, as well as be acted on by the Cut, Copy, Paste, and Delete commands.

The Font Menu

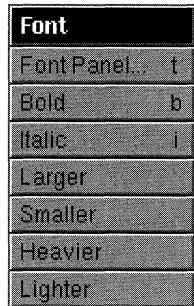


Figure 8-25. The Font Menu

This menu contains the standard Font menu commands and lets you set the font of the text that's currently selected within your application. Interface Builder's use of the Font menu is entirely standard; however, two points are worth mentioning:

- If you enlarge the font used to title an object, the object grows to accommodate the text. If you change the title's font to a smaller one, however, the object's size is not reduced. (Choose the Size to Fit command in the Layout menu to shrink the object so it just accommodates its text.)
- Placing an insertion point in a ScrollView (an object in the Views display of the Palettes window) and then setting the font determines which font the ScrollView will display when a user enters text when the application is run.

The Windows Menu

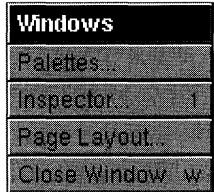


Figure 8-26. The Windows Menu

Palettes

The Palettes command opens a window that presents, in three separate displays, the user interface objects defined by the Application Kit. The three displays are the Views, Windows, and Menus displays and are accessed by clicking the appropriate button at the top of the Palettes window. Dragging an object from the Palettes window to an appropriate destination in your application adds the object to your application's interface file. See "The Palettes Window" below for more information.

Inspector

The Inspector command opens a window that presents, in five separate displays, the characteristics of a selected object. The Inspector window's pop-up list gives you access to these five displays, which are:

Display	Function
Attributes	Lets you examine and modify several graphic attributes (such as an object's title position and border type) and nongraphic attributes (such as a Slider's maximum and minimum values) of the selected object.
Connections	Lets you examine and modify the connections between objects.
Autosizing	Lets you set which (if any) of an object's dimensions will change when the object's superview is resized.
Miscellaneous	Lets you examine and modify the numeric values of the object's frame instance variable. This give you more precise control of an object's size and location than you could achieve by manipulating it with the mouse. This display also shows the name that Interface Builder has assigned to the object.
Class	Displays the Class Inspector for the class of the selected object.

The pop-up list also give you access to one additional display that lets you review and edit the project the interface file is part of:

Display	Function
Project	Displays the Project Inspector. If no project exists, the Project Inspector offers to create one.

See “The Inspector Window” below for more information.

Page Layout

The Page Layout command brings up a standard Page Layout panel. This panel lets you specify how the windows you print using the Print command in Interface Builder’s main menu appear on paper. (See the *NeXT User’s Reference* manual for more information on the standard Page Layout panel.)

Since a screen pixel is approximately 75% of the size of a printer pixel, the image of a window appears larger on paper than it does on the screen. To compensate, set the scaling factor to 75 percent in the Page Layout panel’s Scale field.

All windows are printed in Landscape orientation.

Close Window

The Close Window command closes the key window. The key window may be one of Interface Builder’s windows (such as the Classes or Sounds window), or it may be one of the windows in the application you’re building. To reopen an Interface Builder window, use the Windows menu; to reopen an window in your application, use the File window.

The Layout Menu

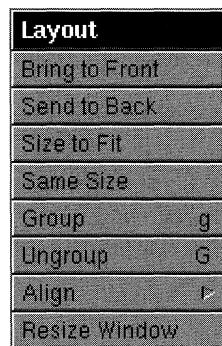


Figure 8-27. The Layout Menu

Bring to Front

The Bring to Front command establishes the selected object as the frontmost object in the window. If the selected object intersects other objects, the selected one is drawn over the other ones; otherwise, no change appears to occur. However, if you bring an isolated object to the front and then drag it so that it intersects other objects, the previously isolated object is drawn over the other objects.

If more than one object is selected when you choose the Bring to Front command, the entire group of selected objects is brought above all other objects in the window.

Send to Back

The Send to Back command puts the selected object behind all other objects in the window. This command is similar to Bring to Front (see previous section) except that the direction is reversed.

Size to Fit

The Size to Fit command resizes the selected object to the minimum size required to display its contents. If more than one object is selected, each is resized to its own minimum size. For an object of a given class, minimum size may depend on the font used to display the title, the alignment and location of the title, and the distance the content area is offset from other areas of the object. The Font menu controls the title's font, and the Inspector window lets you set the other attributes that affect an object's minimum size.

This command has no effect on Sliders, CustomViews, or ScrollViews, or on the length of Forms.

Same Size

The Same Size command forces one or more selected objects to assume the dimensions of another selected object. The first object you select establishes the dimensions that the other selected objects will assume.

Group

The Group command groups one or more selected objects by surrounding them with a titled box. The box is sized so that it just accommodates the objects in the group. Groupings are often used within panels to organize the display of similar items.

The grouped objects become the subviews of the Box object that contains them: Dragging the box drags the grouped objects, and cutting or copying the box cuts or copies the grouped objects. A single click selects the Box, and a double-click (within the area of the box, but not on a grouped object) selects the content view of the Box. Once the content view is selected, you can select and operate on the individual objects in the grouping.

You can group individual objects (such as Buttons or TextFields), previously grouped objects, or a mixture of the two. If you create multiple layers of groupings, you can still access the objects within the innermost group by repeatedly applying the selection technique just described.

Ungroup

The Ungroup command removes the box that groups one or more objects. This command has no effect unless the selection consists solely of a Box object. After an Ungroup command, the objects the box grouped exist as separate Views within the window; they are no longer subviews of the box.

Align

The Align command opens a menu that menu offers several tools to help you position objects within your application's windows. See "The Align Menu" below for more information.

Resize Window

The Resize Window command puts a window in resizing mode, making even windows that won't be resizable at run time temporarily resizable. This command affects only standard windows and panels; menus automatically resize to fit their contents and can't be resized directly.

During application development with Interface Builder, windows that will be resizable at run time display a resize bar. To establish the run-time size of such a window, you simply adjust it using the resize bar. Windows that won't be resizable at run time don't display this resize bar. The Resize Window command lets you adjust the size of such windows by making them temporarily resizable.

When you choose the Resize Window command (or click the resize button in a window's title bar), the window's resize button highlights and the window can be resized. For windows that won't be resizable at run time, a resize bar temporarily appears so that you can adjust the window's size. After you resize the window (or click anywhere within it) the resize bar disappears.

The Align Menu



Figure 8-28. The Align Menu

Alignment

This command opens the panel shown below.

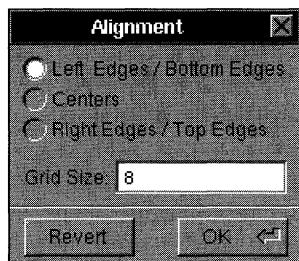


Figure 8-29. The Alignment Panel

The three radio buttons let you specify whether an object's lower left corner, center, or upper right corner is used as the reference point for arranging the object in a row or column of objects. (See "Make Row" and "Make Column" below.) The Grid Size field displays the value used for the vertical and horizontal spacing of the alignment grid. (See "Set Grid On or Set Grid Off" below.) This value must be an integer in the range from 4 to 32.

Make Row

Make Row aligns the selected objects horizontally. The row extends to the right of the object that's nearest the top left corner of the window. The spacing between objects is determined by the original horizontal spacing between the two objects nearest the window's top left corner. If these objects originally overlapped, the objects in the resulting row abut each other.

Make Column

Make Column aligns the selected objects vertically. The column forms below the object that's nearest the top left corner of the window. The spacing between objects is determined by the original vertical spacing between the two objects nearest the window's top left corner. If these objects originally overlapped, the objects in the resulting column abut each other.

Set Grid On or Set Grid Off

Set Grid On enables and Set Grid Off disables the alignment grid in all panels and standard windows of all open interface files. When the grid is enabled, View objects dragged to a window are constrained in their location and dimensions to the units defined by the grid. The intersections of the grid are aligned, both vertically and horizontally, on every eighth pixel in a window. The grid can be made visible by clicking the Show Grid command; see the next section for details.

When you drag an object into a window, the object's lower left corner is the reference point for alignment with the grid. (However, the "Alignment" section above describes how to change this reference point.) As you drag the object within the window, it moves in increments as its lower left corner jumps from intersection to intersection. If multiple objects are selected and moved at once, alignment is set by the lower left corner of the object containing the cursor. Resizing an object when the grid is on causes the resized portion to align with the grid.

Setting the grid on has no immediate effect on objects placed in the window when the grid was off. Their location and dimensions are unchanged until you attempt to move or resize them.

Show Grid or Hide Grid

Show Grid displays and Hide Grid hides the alignment grid in all panels and standard windows of all open interface files. The grid is displayed as a rectangular array of dark gray dots (actually, single pixels) aligned both vertically and horizontally on every eighth pixel. The grid display is an alignment tool to use when you're building your application's interface; it won't appear in the application's windows at run time.

Hiding the grid doesn't turn it off, but showing it will turn it on if it's not on already.

The File Window

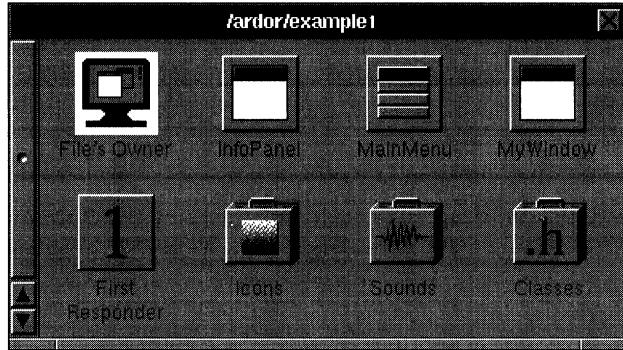


Figure 8-30. The File Window

The File window gives you a top-level view of the objects and resources that make up your application. It contains icons that represent:

- The interface file's owner
- The first responder
- A font manager object, if your application uses the Font panel
- Your application's icon, sound, and class resources

Icons that represent objects within your application are square. You can use one of these icons as the source or destination of connections (see “The Connections Display” below) and to recall the actual object to the screen. For example, to display an application’s Info panel, double-click the InfoPanel icon in the File window.

The icons for editable objects have black titles; those that aren’t editable have gray titles. You can cut, copy, and paste the icon for an editable object. Doing so performs the same operation on the object the icon represents. Icons that represent the interface file’s owner or the first responder can’t be edited in any way.

You can also change the name of an editable object by double-clicking the title that’s displayed below its icon. Note, however, that the object’s name has little bearing on the work you do in your application: These names are primarily for Interface Builder’s use. To keep track of the objects in your application, Interface Builder uses **NXNameObject()** to construct an association between the object, a name, and an owner object for each object in the interface file. (See **NXGetNamedObject()** in *NeXTstep Reference, Volume 2*, for more information on the functions that associate objects, owners, and names.) Although it’s generally easier for your application to access an object in an interface file through an outlet of the owner object, your application could access the object by calling **NXGetNamedObject()** with the object’s name and owner as arguments. The names of your application’s top-level objects (windows, menus, owner, and so on) are displayed in the File window. To find out the names of lower-level objects (such as the buttons and text fields in a window), use the Miscellaneous display of the Inspector window, described later in this chapter.

Icons that represent resources look like suitcases. Double-clicking a resource icon opens a window that displays the resources of that type. Opening the Icons window, for example, displays a collection of icons that you can assign to certain objects in your application. You could drag the Return icon from the Icons window to a button in your application. When you release the mouse button, the icon is assigned to the button. You assign sounds to buttons in the same way.

The title bar of the File window displays the name and directory location of the interface file. By default, the first interface file you create during an Interface Builder session is called **Untitled1**. You can change the name when you save the file. Each new interface file you open opens its own File window. Closing a File window closes the interface file that's associated with it. If there are any outstanding changes, a panel will prompt you to save the file before you close it.

File's Owner

The icon titled "File's Owner" represents the owner of the interface file. The owner is an object outside the interface file that serves as an intermediary between the objects unarchived from the interface file and the other objects of your application. In general, your application accesses the unarchived objects through the outlets of the owner object. In turn, the unarchived objects access the other objects in your application by sending messages to the owner object.

The owner must exist before any of the objects specified in its interface file can be created. For example, the interface objects from the **example1.nib** interface file might be hooked to their owner object (of the fictitious MasterObject class) in this way:

```
theOwner = [MasterObject new];
[NXApp loadNibSection:@"example1.nib" owner:theOwner];
```

Before the **loadNibSection:owner:** method returns, the objects are unarchived from the interface named section of the executable file, and the connections you specified within Interface Builder are established. For example, those objects that send actions messages to a target are initialized with the **ids** of their targets. The **ids** of the objects unarchived from the interface file are known within the scope of the **loadNibSection:owner:** method, and so it's possible to initialize **target** instance variables to any of these **ids**. Since the owner's **id** is also known within the scope of the **loadNibSection:owner:** method (it's the method's second argument), the owner can be made the target of action messages from unarchived objects. In fact, it's the only object outside the interface file that can be the target of action messages from unarchived objects.

In the same way, if within Interface Builder you've connected the owner's outlets to objects within the interface file, these outlets are initialized to the proper **ids** when the **loadNibSection:owner:** method is applied. For this to happen, the owner must have

separate methods to set each of its outlet variables. An outlet named **windowOneOutlet**, for example, requires a method of this format:

```
- setWindowOneOutlet:anObject
{
    windowOneOutlet = anObject;
    return self;
}
```

Warning: The capitalization of the method name is critical. For each outlet variable of the format *wordWord*, the Application Kit will send a *setWordWord:* message. Unless such a method is provided, an error occurs. If you let Interface Builder create a template file for all custom classes, these methods are written for you. (See “The Class Inspector” for more information.)

First Responder

The First Responder icon represents the object within a window that will be the first to receive keyboard events, mouse-moved events, and action messages from Controls that don’t have an explicit targets. (See “The First Responder” in Chapter 7 for a full discussion of first responders.)

In most cases, a window’s first responder will be chosen from the window’s Text objects (or objects that use Text objects such as Form, TextField, and ScrollView objects). Clicking one of these objects generally makes it that window’s first responder. Over time, many different objects can become the first responder, but at any one time, only one object has this status. The First Responder icon stands for the object that has this status, no matter which actual object it is within your application.

Having First Responder in the File window lets you connect an object, such as the MenuCell that sends the **copy:** message, so that it sends its action message to a target whose identity changes over time. Thus, for example, the Copy command can be set up to work with any TextField in a window, as long as the TextField is the first responder. (Incidentally, Objective-C’s ability to let the target of a message be defined at run time rather than at compile time is an example of dynamic binding. For more information on Objective-C, see Chapter 3.)

Font Manager

A FontManager object appears in the File window only if you’ve dragged a Font menu item from the Palettes window into your application’s main menu. The FontManager receives action messages from the MenuCells in the Font menu. The connections between the Font menu and the FontManager are already established for you. The FontManager, in turn, works with the first responder to convert a font or fonts in the current text selection.

Icons

The resource icon titled “Icons” in the File window represents the icons available in your application’s interface file. When you double-click the resource icon, the Icon window opens.

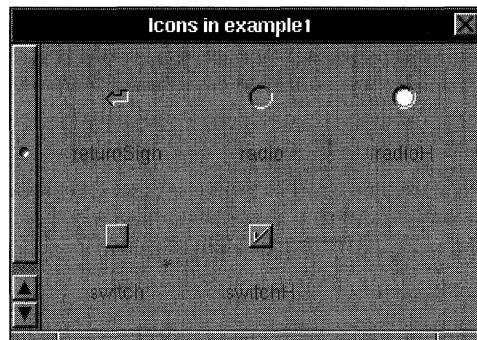


Figure 8-31. The Icons Window

You can drag an icon from this window into one of your application’s windows and deposit the icon on a Button or ButtonCell object of your choice. To indicate that an object has been selected as a destination, a black outline appears around it. If you release the mouse button while the black outline is displayed around an object, the icon is assigned to that object.

When you first start Interface Builder, the icons you see in the Icons window are a collection of system icons. These are icons used by the Application Kit and so can’t be edited or renamed. However, you can add icons to the Icon window in either of two ways:

- You can locate an TIFF or EPS file in one of the Workspace Manager’s directory windows and drag its file icon into the Icon window.
- You can create an icon using an application such as Icon (located in **/NextDeveloper/Demos**), copy the icon onto the pasteboard, and then paste it into Interface Builder’s Icon window.

Interface Builder uses the image information from one of these sources to create a Bitmap object that it archives in the application’s interface file. The Bitmap object appears in the Icons window. Once they’re displayed in the Icon window, these icons can be cut, copied, and pasted, and their names can be changed.

If the width or height of the icon you add to the Icons window exceeds 48 units (in the base coordinate system), it won’t be displayed directly. Instead, a generic icon takes its place. To see the actual image, select the generic icon and open the Inspector window. See “Icon Attributes” below for more information.

Sounds

The Sounds icon in the File window represents the sound effect resources available in your interface file. Double-clicking the Sounds resource icon opens the Sounds window.

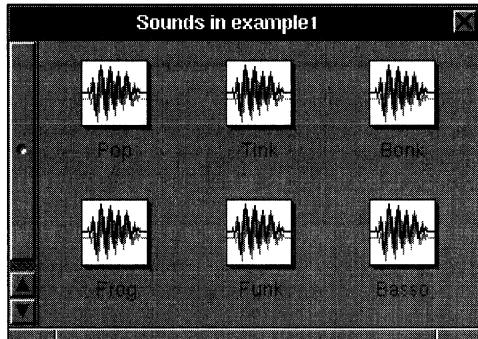


Figure 8-32. The Sounds Window

Each icon in the Sounds window represents a sound file; the icons you see when you first open the Sounds window represent a selection of system sounds. To add a sound to the Sounds window, drag the sound file icon from a directory window to the Sounds window. For a sound you add, the title of the file icon will be displayed in black, indicating the file name can be edited. The title of system sounds is displayed in gray, indicating that these file names can't be edited.

When you add a sound to the Sounds window, the sounds data is actually copied into the interface file. Within the interface file, sounds that you add can be cut, copied, and pasted, and their names can be changed.

Double-clicking a sound file icon plays that sound. To see the sound's waveform and to get some idea of the sound's duration, you can open the Sound Inspector. See the “Sound Attributes” section below for more information. You assign sounds to objects in your application in the same way as you assign icons. See the previous section for details.

Classes

The Classes resource icon in the File window represents the classes available in your interface file. Double-clicking the Classes resource icon opens the Classes window.

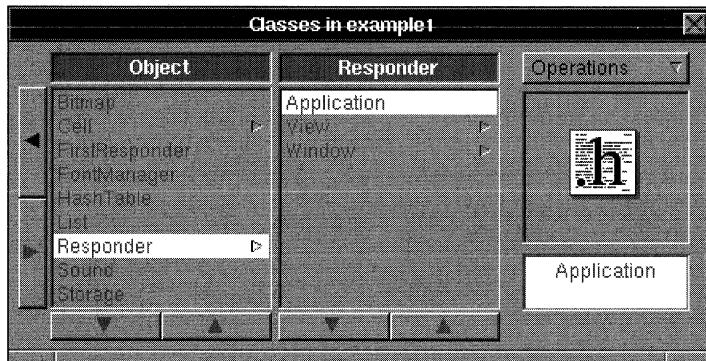


Figure 8-33. The Classes Window

The browser in this window gives you access to the currently defined class hierarchy. Each class is displayed in proper relationship to the other classes: A class's superclass is displayed to its left and its subclass is displayed to its right. Application Kit classes are displayed in gray, indicating that they can't be edited. The classes you define, being editable, are displayed in black.

The icon well contains the icon for the interface of the selected class. If you double-click the icon (or the class name in the browser), two things occur. First, the Class Inspector appears and displays the class's outlets and action methods (if any). Second, if the class's interface file exists in the current directory, an Edit window appears displays the interface file for reference purposes.

You can add classes to this hierarchy in two ways:

- Drag to the Subclass button in the pull-down list. This button creates a new class as the subclass of the class that's currently selected in the browser. New classes are named Subclass1, Subclass2, and so on.
- Drag the icon for a class interface file from the directory browser to the icon well in the Classes window or to the File window. When you drag the icon to either of these destinations, the Classes resource icon in the File window opens to accept the interface definition. Interface Builder parses the interface file and places the new class in its proper place in the class hierarchy.

In either case, the new class interface isn't written to a file in the current directory. To do this, you must choose the Unparse button in the Operations list. Each of the buttons in this list are described below.

Parse

The Parse button reads the class definition for the selected class from an interface file on the disk. Interface Builder looks for the class interface file in the current directory. If it finds the file, Interface Builder reads the interface data from the file and then displays the name of the class in its proper location in the browser.

Using the Parse button implies that you've previously specified the class's name and position in the class hierarchy. (See the Subclass button to learn how to do this.) Only then can you parse the class's interface file. A more convenient way to add an existing class to your application is to drag the file icon for the interface file directly into the Classes window (or the File window) so that Interface Builder can automatically parse the file and place the class in the class browser.

Unparse

The Unparse button generates class files for the selected class to the current directory. It writes a complete interface file based on the outlets and action methods you defined for the class using the Class Inspector. It writes a skeletal implementation file, providing methods that set each of the class's outlets and skeletal methods for each of the class's action methods.

- Warning:** If you edit the files Interface Builder generates, Unparse can overwrite the edited files with new class files. If source files already exist when you choose Unparse, Interface Builder asks for verification before overwriting them.

Subclass

The Subclass button defines a subclass of the class that's currently selected in the browser. The subclass is given a default name (as described above) that you can change using the text field below the icon well. Once you've set the class's name and position in the class hierarchy, you can open the Class Inspector to define its outlets and action methods.

Instantiate

The Instantiate button creates an object of the selected class and places an icon representing that object in the File window. The object's name refers to its class. For example, if you define the Gauge class and then choose Instantiate, the object that appears in the File window is named GaugeInstance. Having the icon in the File window lets you create connections between this object and other objects. This button is disabled for View classes since it's more convenient to instantiate a View object by dragging a CustomView from the Palettes window. By using this technique, you not only instantiate the View object, but you can set its position and size within your application's window.

The Palettes Window

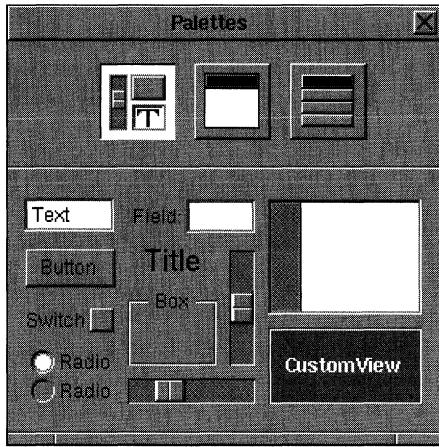


Figure 8-34. The Palettes Window

This window gives you access to the Application Kit's primary user interface objects. The objects are divided into three groups, which are accessed by the three buttons at the top of the window. The left button opens the View palette (shown above), the middle button the Window palette, and the right button the Menu palette.

To add an object from the Palettes window to the current interface file, simply drag the object to an appropriate destination and release the mouse button. The appropriate destination is determined by the type of object:

Class of Object	Appropriate Destination
View	Anywhere within a Window or Panel in your application's interface
Window	Anywhere in the workspace
MenuCell	Anywhere within a menu or submenu in your application's interface

Once you place an object in your application, you can edit it either by direct mouse actions (such as dragging a control point to resize the object) or by using the Inspector window. Besides identifying the individual objects, the following sections point out what direct manipulations are possible beyond simple resizing and repositioning. See the section "The Inspector Window" for a discussion of what indirect manipulations are possible for each object.

The View Palette

This display offers a palette of View objects in configurations that most programmers will find convenient. If two or more configurations are commonly used (such as the horizontal and vertical orientations of the Slider), the View palette offers an example of each. These objects are only a starting point; you can always customize an object taken from the Palettes window by editing it using the Inspector window.



TextField objects. The first is configured for editable text and the second for static text displays. Once placed in a window, either of these objects can be Alternate-dragged into a matrix of identical objects.



Button objects. The first is a momentary push button and the second is a toggle. Once placed in a window, either of these objects can be Alternate-dragged into a matrix of identical objects.



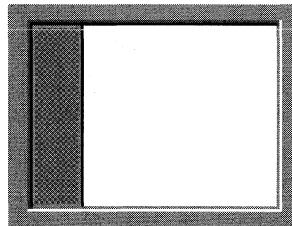
Matrix of ButtonCell objects. The ButtonCells are configured as toggles, and the Matrix is in radio mode, which allows only one cell of the matrix to be selected at a time. You can alter the number of cells in the matrix by Alternate-dragging. Double-clicking a ButtonCell within the Matrix selects the ButtonCell for further editing with the Inspector window.



Form object containing one FormCell. By Alternate-dragging the middle control point at the top or bottom of this object, you can change the number of cells the Form contains. Double-clicking a FormCell within the Form selects the FormCell for further editing with the Inspector window.



Slider object. The Palettes window offers both vertical and horizontal sliders. Alternate-dragging one of the control points near the center of a slider creates a matrix of identical Sliders.



ScrollView containing a Text object. Once you've dragged this object into one of your application's windows, you can add text to it by double-clicking within the white area of the background. Any text you add will appear when the application is run. If you add more text than the ScrollView's content area can accommodate, the ScrollView's vertical scroller is enabled.



Any subclass of View. This object, a placeholder for an object of your own design, provides a convenient way of setting the size and location of a View object that's unknown to Interface Builder. You use the Inspector window to set the View subclass for this object. For information on adding a custom object that isn't a subclass of View, see "Classes" under "The File Window" above.

The Window Palette

This display offers a palette of Window and Panel objects. When you start a new application, Interface Builder gives you a standard window and an information panel. If your application requires additional windows or panels, you drag them from this palette into the workspace.

Each window or panel in the application is represented by an icon in the File window. You can cut or copy a window or panel by selecting it or its icon and then choosing the appropriate command.

The windows and panels you add will, by default, be objects of the Window and Panel classes. If you want to make a window or panel an instance of another class, you must define the class using the Classes window and then use the Inspector window to assign this class to a specific window or panel.

The Menu Palette

This display offers a palette of menu commands. To add another command to a menu in your application, you simply drag the appropriate one from the Menu palette and deposit it in your application's menu. Once the command is in the menu, it can be dragged vertically within the confines of the menu to reorder it in the menu's list of commands.

The top two commands (Item and Submenu) in the Menu palette are entirely generic; you must label and connect them to the rest of your program. The remaining commands (Edit, Window, and Font) bring up predefined submenus. As described below, some of the commands in these submenus are hooked to the appropriate objects in your application; others you have to connect explicitly. See “The Connections Display” under “The Inspector Window” below for information about connecting objects.



The Window command. This command displays the standard Window menu, which contains these commands:

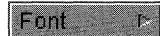
- Open
- New
- Save
- Save As
- Save To
- Revert to Saved
- Miniaturize
- Close

Only the last two commands are connected to your application. Both send messages to the first responder. To connect the other commands, use the Inspector window. The Open, Save, and Close commands have their standard keyboard alternatives.



The Edit command. Since a new application created with Interface Builder will already have the Edit command in its main menu, you'll rarely need this menu item except to restore the Edit command after it has been cut from a menu.

The commands in the Edit menu have been assigned their standard keyboard alternatives and are configured to send their action messages to the first responder.



The Font command. This command displays the standard Font menu, which contains these commands:

- Font Panel
- Bold
- Italic
- Larger
- Smaller
- Heavier
- Lighter

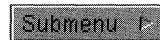
These commands work by sending messages to a FontManager object. Consequently, adding the Font command to your application causes Interface Builder to add a FontManager object to the application's interface file. The icon for this object appears in the File window.

Each command in the Font menu is connected to your application's FontManager. The Font Panel, Bold, and Italic commands have their standard keyboard alternatives.



A generic command. You can edit the title of the menu item by double-clicking the title "Item." To assign a character to the command as a keyboard alternative, first double-click the area to the right of the title to select the area that displays the keyboard alternative and then enter the character. You must explicitly connect the new menu command to a target in your application.

Note: Interface Builder doesn't check that the keyboard alternative you assign for a given command is unique within your application's menus. Also, you should avoid assigning any of the characters that have been reserved for the standard NeXT keyboard alternatives.



A generic submenu command. When you add this command to a menu, a submenu appears that has one generic menu command. The submenu has the same title as that of the menu command. Editing the menu command automatically changes the title of the submenu. You must explicitly connect the command in the submenu to a target in your application.

Although it's rarely needed, you can make a hierarchy of menus by adding a submenu item to an existing submenu.

The Inspector Window

The Inspector window is the primary tool for editing the nongraphic features of the objects in your application. With it, for example, you can set a slider's minimum and maximum values or a window's buffering type. It also lets you set some of the graphic features. The graphic features the Inspector window offers for editing are generally those that are best represented as a series of choices. For example, you set an object's foreground and background gray values by choosing a button in the Inspector window that shows the values you want.

The features that the Inspector window displays for editing depend on which object in your application is currently selected: The Inspector window looks substantially different when a Button is selected than when a Slider is selected. There are, however, two aspects of the Inspector window that remain unchanged no matter which object is selected: the button at the top of the window and the Revert and OK buttons at the bottom of the window.

The button at the top of the window controls a pop-up list. This list lets you switch to any of the Inspector window's five displays that are related to the selected object:

- Attributes
- Connections
- Autosizing
- Miscellaneous
- Class

As a convenience, the list also contains this option that opens the Project Inspector:

- Project

Each of these displays and commands is discussed in a separate section below.

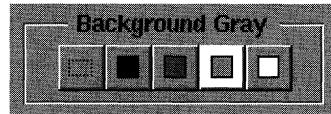
At the bottom of the window are two buttons labeled Revert and OK. The OK button lets you confirm the choices you've made within the window. No changes take effect until you click OK. The Revert button lets you undo any changes you've made within this display since the Inspector window was opened or since the last time the OK button was clicked, whichever happened most recently.

The Attributes Display

More than any of the other three displays, the content of the Attributes display depends largely on the object that's selected in your application. In general, use the Attributes display when you want to edit an object's background or text color, border type, text alignment or editability, or other basic characteristics such as a button's type (momentary push, toggle, and so on).

Since the Attributes display changes depending on the object being edited, the following sections discuss the display on an object-by-object basis.

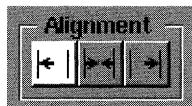
TextField Attributes



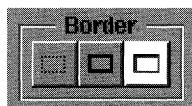
The Background Gray buttons control the gray value for the background of a TextField. From left to right, the choices are: transparent, black, dark gray, light gray, and white. All five choices are only available to TextFields having no border or a black line border. TextFields with bezeled borders can't have a transparent background. (See the Border description below.) By convention, TextFields containing editable text have a white background.



The Text Gray buttons control the gray value of the text displayed in a TextField. Since the legibility of the text depends on the contrast between the text and its background, the choice you make here should take into account the gray value you select for the TextField's background.



The Alignment buttons set the alignment of the text displayed in a TextField. The choices, from left to right, let you specify that text will be aligned with the TextField's left margin, center axis, or right margin. Setting the alignment affects how the text appears while you're building the application and how it will appear at run time.



The Border buttons determine the border type for the TextField. The choices are: no border, a black line border, or a beveled border. By convention, TextFields that contain editable text have beveled borders. Only TextFields having no border or a black line border can have transparent backgrounds. (See the Background Gray description above.)

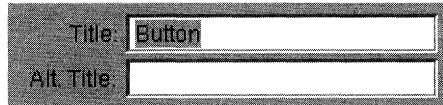


The Option buttons set whether the text within a TextField can be edited, selected, and scrolled in Test mode or at run time. The effects of these options are interrelated. If you specify that the text is editable, the Selectable button will be checked and disabled. In addition, the Scrollable button has effect only on selectable TextFields: TextFields that can't be selected can't be scrolled.

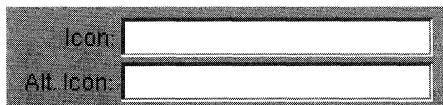


This field sets the TextField's tag. A tag is a number that the TextField's target can use to identify which of several objects sent a particular action message. (See "Tags" in Chapter 6 for more information.)

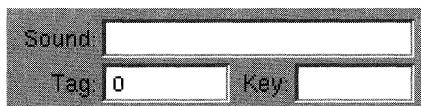
Button Attributes



These fields set the button's title and alternate title. Generally, you set the title directly by editing the text that appears on the button; however, you can also set it here. You can only set the alternate title using the Inspector window. The alternate title is displayed when a momentary-change or toggle button is clicked. If the title or alternate title you specify is wider than the button's title area, the button grows to accommodate the new text.



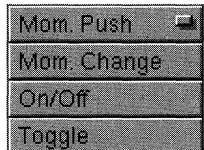
These fields set the button's icon and alternate icon. It's most convenient to specify an icon by dragging it from the Icons window directly to the button. When you do so, the icon's name appears in the field in the figure above. Alternate icons must be specified by name, using this portion of the Inspector window.



The Sound field sets the Button's sound. Although you can set the sound associated with a button by entering the pathname of a sound file, it's generally more convenient to drag a sound icon from the Sounds window directly to the button. (See the preceding "Sounds" section for more information.)

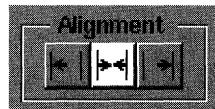
The Tag field sets the Button's tag. A tag is a number that the Button's target can use to identify which of several objects sent a particular action message. (See "Tags" in Chapter 6 for more information.)

The Key field sets the character that's used as the Button's keyboard alternative.

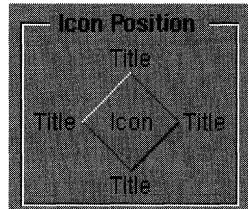


This pop-up list sets the Button's type.

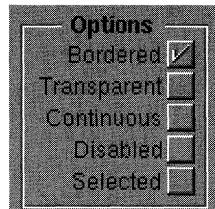
Type	Description
Momentary Push	A button that stays highlighted as long as it's pushed. If it's bordered, it also pushes in when clicked. It doesn't display an alternate title or icon.
Momentary Change	A button that displays its alternate title and icon (if any) as long as it's pressed. It doesn't push in or highlight when clicked.
On/Off	A button that highlights when clicked and then retains its highlight until being clicked again. If it's bordered, it also pushes in when clicked. It doesn't display an alternate title or icon.
Toggle	A button that displays its alternate title and icon (if any) when clicked and then continues to display these characteristics until it's clicked again. If it's bordered, it also pushes in when clicked. It doesn't highlight.



The Alignment buttons set the alignment of both the title and alternate title within the button's title area. The left button aligns the text with the left edge of the title area, the middle button centers it within the title area, and the right button aligns it with the right edge. If the button has an icon to the left or right of the title, the title area is reduced by the width of the icon.



The Icon Position buttons set the relative position of the button's icon and title. The diamond-shaped button's title can read "Title" or "Icon." If it says "Title," the button is disabled; if it says "Icon," the button is enabled. When the button is enabled, you can click any point of the diamond to specify the position of the icon in relation to the title.



The Options buttons set various characteristics.

Option	Effect When Selected
Bordered	Gives a button a raised border. Buttons have this appearance by default.
Transparent	Makes a button invisible by disabling its drawing and letting the button's background show through. Despite its transparency, the button still responds to mouse actions and so can be used to sensitize areas of a window. In Interface Builder, a transparent button is indicated by a white rectangle; at run time, the button will be truly transparent. Although a transparent button isn't visible, it still sends its action message to its target when the user clicks within the button's bounds.
Continuous	Specifies that for as long as a button is pressed, it will send repeated action messages to its target. By default, when a continuous button is pressed, it waits .2 seconds before sending action messages at .025 second intervals. You can alter the delay before the first message is sent and the frequency of the messages that follow, with ButtonCell's setPeriodicDelay:andInterval: method.
Disabled	Sets a button's state: either enabled or disabled. The letters of a disabled button's title are dark gray rather than black.
Selected	Sets the initial state of a two-state button (an On/Off or Toggle button) to its second state.

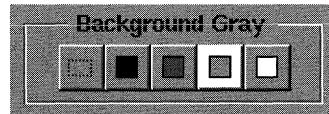
Matrix Attributes

A Matrix is a Control that comprises one or more objects of the Cell class. Because it's an object that groups—and helps coordinate the actions of—other objects, you need a way to inspect both the Matrix and the Cells within it.

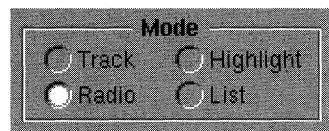
A single click within a Matrix selects it and, if the Inspector window is open, displays the Matrix Inspector. A double-click within a Matrix selects a Cell in it and displays the appropriate inspector for that Cell. The inspectors for individual Cell classes are identical to inspectors described elsewhere in this section:

Cell Class	See Inspector For
TextFieldCell	TextField
ButtonCell	Button
SliderCell	Slider

The inspector for the Matrix itself is described in this section.

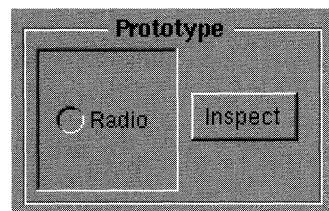


The Background Gray buttons control the gray value and transparency of the Matrix's background, the area that surrounds the Cells and is visible behind transparent Cells. From left to right, the choices are: transparent, black, dark gray, light gray, and white. You can also set the background for any or all of the Cells within the Matrix by inspecting the Cells themselves.



The Mode buttons set the mode of operation for the Matrix.

Mode	Description
Track	When the mouse button goes down within the Matrix, the Cells track the cursor. No highlighting occurs. A Matrix of SliderCells uses this mode.
Highlight	When the mouse button goes down within the Matrix, the Cells track the cursor. Each Cell is highlighted whenever the cursor is within its rectangular area.
Radio	When the mouse button goes down within the Matrix, the Cells track the cursor. Releasing the mouse button when the cursor is within the rectangular area of a Cell selects the Cell. Radio mode allows no more than one Cell to be selected at a time.
List	List mode is similar to radio mode except that list mode allows multiple Cells to be selected. In list mode, you can select multiple Cells by dragging, extend the selection by Alternate-clicking, and create discontinuous selections by Shift-clicking.

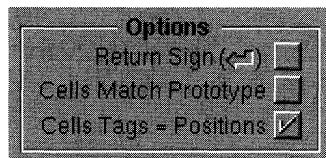


The Inspect button in the Prototype group lets you modify the attributes of the prototype Cell of the Matrix. The inset box displays the current prototype Cell.

By Alternate-dragging a control point of a Matrix, you can enlarge the Matrix and add Cells. The class and attributes of the new Cells are determined by the prototype Cell of the Matrix.

The icon well displays the current prototype Cell for the selected Matrix. To change the attributes of the prototype, click the Inspect button. The display that appears is for the Cell class of the prototype Cell. Once you've chosen the attributes you want for the prototype and clicked OK in the Inspector window, the Matrix Inspector display reappears.

A Matrix that you create with Interface Builder can contain Cells of only one class (a Matrix of ButtonCells, for example) but the Cells within the Matrix can have various attributes (some of the ButtonCells can display borders while others don't). Few applications will need a Matrix composed of Cells of different classes, so Interface Builder doesn't support the creation of such a Matrix. If your application requires one, you'll have to use the methods provided by the Application Kit.



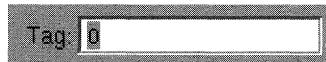
The Options buttons set various characteristics of the Cells in a Matrix.

The Return Sign button has two effects. When it's checked, the ButtonCell that's nearest the lower right corner of the Matrix (in a Matrix of ButtonCells) displays the Return icon. When the user presses the Return key at run time (or in test mode), this ButtonCell responds as if it had been clicked with the mouse. The Return Sign button is enabled only if the selected Matrix contains ButtonCells.

The second button in this group determines whether the Cells in the Matrix will conform to the properties of the prototype Cell. When you drag out a Matrix of objects, they will be of one type. However, you can create a Matrix having Cells of various attributes by changing the prototype Cell and then adding Cells to the Matrix. If you later decide to make all Cells in the Matrix conform to a single set of attributes, set the prototype cell and then check the Cells Match Prototype button. After you click OK, the Cells in the Matrix will all conform to the prototype's attributes.

The last button in this group lets you set the tag of each Cell in a Matrix equal to the Cell's positional value within the Matrix. If you check this button and then click OK, the first Cell's tag equals 0, the second equals 1, the third 2, and so on.

The choices you make with these buttons affect the Matrix only until the next time you change the number of its Cells. If you add or subtract Cells from the Matrix, you'll have to reestablish these settings.

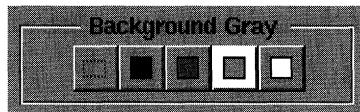


This field lets you set the tag value for the Matrix. A tag is a number that the target of an action message can use to identify which of several Controls sent a particular action message.

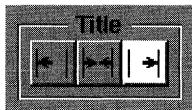
You can specify a tag value for each of the Cells within the Matrix as well as for the Matrix itself. By default, all Cells you add to a Matrix have a tag value of 0. Selecting a Cell within the Matrix displays the inspector for that type of Cell and lets you change the Cell's tag, among other things. Using the Matrix Inspector, you can set each Cell's tag equal to its position in the Matrix, as described in the previous section.

Form Attributes

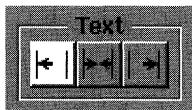
A Form is a Matrix of FormCells. The Form provided in the Palettes window has just one FormCell; once you drag a Form into a window, you can Alternate-drag the Form so that it contains any number of FormCells. Since it's a Matrix, a Form shares many of the same attributes, as the descriptions below illustrate.



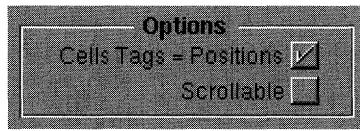
The Background Gray buttons control the gray value and transparency of the Form object's background. The background consists of the area that surrounds each FormCell and that's visible behind transparent FormCells. From left to right, the choices are: transparent, black, dark gray, light gray, and white.



The Title buttons set the alignment of the text in the title area of the Form's FormCells. The left button aligns the text with the left edge of the title area, the middle button centers it within the title area, and the right button aligns it with the right margin.



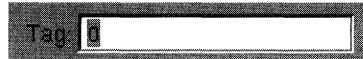
The Text buttons set the alignment of the text in the text field area of all the Form's FormCells. The left button aligns the text with the left edge of the field, the middle button centers it within the field, and the right button aligns it with the right edge.



The Options buttons set whether the tag value for each FormCell is equal to its positional value in the Form and whether the text displayed in the Forms text fields is scrollable. These options apply equally to all FormCells in the Form; for example, checking the Scrollable button makes all of a Form's FormCells scrollable.

The first button of this pair lets you set the tag of each FormCell in a Form equal to the FormCell's position within the Form. By checking this button and then clicking OK, the first Cell's tag equals 0, the second equals 1, the third 2, and so on.

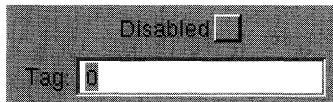
The second button sets whether text that's entered in any of the Form's text fields will be scrollable. Scrolling can occur only when a text field holds more text than can be displayed within its area. The user scrolls the text either by dragging the selection beyond the left or right edge of the text area or by entering additional text in the text field.



This field lets you set the Form's tag. (A tag is a number that a control's target can use to identify which of several Controls sent a particular action message—see “Tags” in Chapter 6 for more information.) By selecting a FormCell within the Form and displaying the FormCell Inspector, you can set the tags of the individual FormCells.

FormCell Attributes

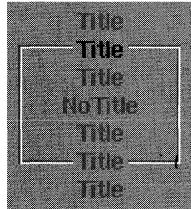
FormCells are the building blocks of a Form object. A Form can contain one or more FormCells. By double-clicking a Form, you select one of the FormCells within it. The FormCell Inspector lets you set two important attributes of the selected FormCell.



The Disabled button sets whether the FormCell will respond to input from the user. The text in a disabled FormCell's title and text field areas is dimmed and the text in the field cannot be selected or edited. If the user presses Tab or Shift-Tab to move the insertion point from field to field, the disabled fields are skipped.

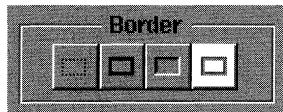
The Tag text field sets the FormCell's tag. (A tag is a number that the control's target can use to identify which of several controls sent a particular action message—see “Tags” in Chapter 6 for more information.)

Box Attributes

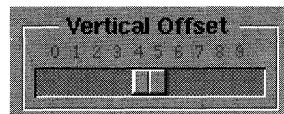
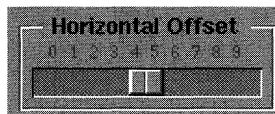


These borderless buttons set the location of the box's title relative to its border. By default, the title intersects the border at the top of the box, as indicated by the darkened choice near the top of this display. By clicking any of the other choices, you can change the location of the title for the selected box. Clicking the "No Title" choice removes the display of the title.

Changing the title's location doesn't affect the box's frame rectangle but, depending on the new location of the title, may change the box's content area.



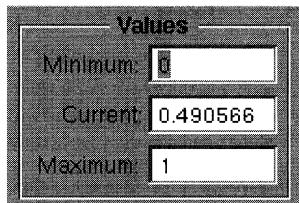
The Border buttons control the type of border the box displays. The four buttons, from left to right, correspond to these types: no border, a black line, a bezel, and a ridge. When you change the border type, the box's frame rectangle is resized to keep the content area constant.



These sliders set the horizontal and vertical distances that the box's border is offset from its content area. The horizontal offset is applied equally to the left and right of the content area, and the vertical offset is applied equally above and below the content area. For example, the border could be offset five pixels to the left and right of the content area and three pixels above and below it.

When you change a box's offsets, the content area is held constant and the frame rectangle is resized to accommodate the new offset dimensions.

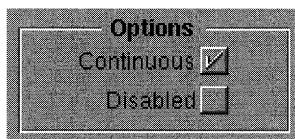
Slider Attributes



The Values fields let you set the minimum, maximum, and current values of the selected slider. The maximum value is represented by the right end of horizontal sliders and the top end of vertical sliders. The current value is represented by the placement of the knob.

The values you enter in these fields can be signed or unsigned integers, signed or unsigned floating-point numbers, or either of these followed by an exponent. For example, any of these numbers would be acceptable:

-100
-100.0
-1.0e2



The Options buttons set how a slider responds to user actions: whether it sends action messages continuously when the knob is dragged and whether it's enabled or disabled.

If the Continuous button is checked, the slider sends action messages to its target continuously as the knob is dragged; otherwise, the slider sends an action message only when the user releases the mouse button after manipulating the slider.

If the Disabled button is checked, the slider is disabled and won't respond to the user's actions. Disabled sliders display a light gray bar in contrast to the dark gray bar of enabled sliders.

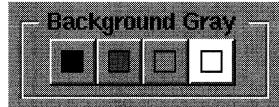


This field lets you set the Slider's tag. A tag is a number that the control's target can use to identify which of several Controls sent a particular action message. (See "Tags" in Chapter 6 for more information.)

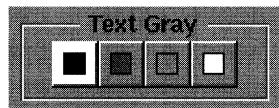
ScrollView Attributes

The ScrollView offered in Interface Builder's Palettes window is configured for displaying text: The ScrollView's document view is a Text object. Except for the Border options described below, the choices in the Attributes display actually configure the Text object the

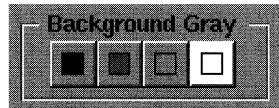
ScrollView contains. If you need to set Text object attributes that aren't listed here, your program must query the ScrollView for the **id** of the Text object and then send messages directly to it.



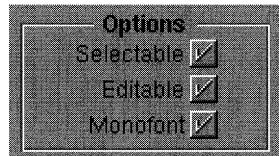
The Background Gray buttons set the gray value for the background of the ScrollView. From left to right, the choices are: black, dark gray, light gray, and white. By convention, ScrollViews containing editable text have a white background.



The Text Gray buttons set the gray value of the text displayed in a ScrollView. Since the legibility of the text depends on the contrast between the text and its background, the choice you make here should take into account the gray value you select for the ScrollView's background. By convention, editable text is black.



The Border buttons determine the border type for the ScrollView. The choices are: no border, a black line border, or a bezeled border. Changing the border type has no effect on the ScrollView's frame rectangle but, depending on the new border, may change the dimensions of the ScrollView's content area.



The Options buttons set the selectability, editability, and font characteristics of the ScrollView's text.

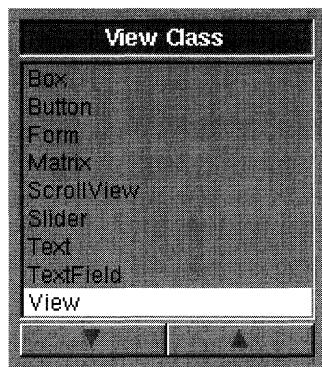
The Selectable and Editable buttons are interrelated. Editable text must be selectable; however, the converse isn't true. You can specify that the text is selectable but not editable. In this way, the user can copy the selection but not alter it.

The Monofont button sets how a ScrollView treats font information that accompanies the text on the pasteboard. If Monofont is checked, font information on the pasteboard is ignored, and the pasted text assumes the ScrollView's current font. Otherwise, the pasted text is displayed using the font information found on the pasteboard.

The Selectable and Editable buttons have no effect on the way the ScrollView operates while you're building the application: They only affect the ScrollView during testing or at run time. In contrast, the Monofont button affects the ScrollView in build mode as well as during testing and at run time.

CustomView Attributes

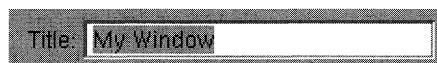
The CustomView object in the Palettes window is a proxy for a View object whose precise class you assign later. After you drag a CustomView object into one of your application's windows, you can use this display to assign it a class.



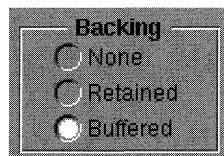
This browser displays the names of the classes available for the CustomView. The View classes of the Application Kit are shown as well as any View subclasses you've previously defined. To assign the CustomView to one of the displayed classes, select the class name in the browser and click OK. The CustomView's name changes to that of the class.

Use the Classes window to add classes to the list displayed in this browser.

Window Attributes

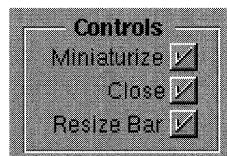


This field sets the title of the window. The title is displayed centered in the window's title area. If the title is longer than the window can accommodate, only the leftmost words of the title are displayed. If you resize the window and increase its width, additional words are brought into view.



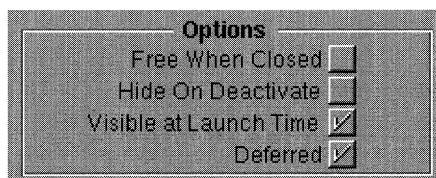
The Backing buttons specify the window's buffering type. The None button specifies that the window has no off-screen buffer: All drawing is done directly to the screen. A retained

window has an off-screen buffer: Drawing is done directly to the screen for visible parts of the window and to the buffer for parts of the window that are off-screen or covered by other windows. A Buffered window has an off-screen buffer that receives all drawing before the drawing is flushed to the screen. See “Window Buffering” in Chapter 4 for more information.



These buttons set which window controls—miniaturize button, close button, and resize bar—a window will display when the application is run.

While you’re creating an application in Interface Builder, each standard window displays a resize button and a close button. These controls allow you to manipulate a window while you’re building the application. However, they have no bearing on which controls the window will display at run time (or during test mode). To specify a window’s run-time controls, click the appropriate buttons in the Controls group.



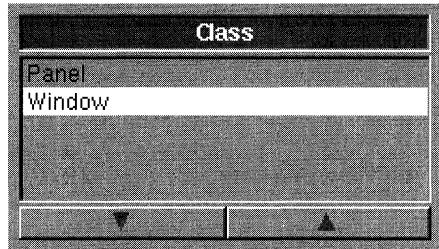
The Options buttons set miscellaneous window characteristics.

The Free When Closed button controls whether the memory the Window Server allocates for a window is freed whenever the window is removed from the screen. You can reduce your application’s memory requirements by specifying that rarely used windows are freed whenever the user closes them.

The Hide On Deactivate button sets whether the window will disappear from the screen when the user activates another application. By convention, an application’s panels are hidden when an application is deactivated but its standard windows aren’t. Interface Builder supports this convention through the settings of this button: By default, the button is checked for panels but not for standard windows.

The Visible at Launch Time button specifies whether the window will appear on the screen when the application starts. Standard windows you add to your application will, by default, be visible when the application starts; panels won’t.

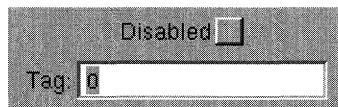
If Deferred is checked, the Window Server doesn’t allocate memory for the window until the window is brought on screen with the **orderFront:** or **orderWindow:relativeTo:** methods. By deferring the allocation of memory for windows that may be accessed rarely or not at all (such as the windows of a help system), you can reduce your application’s startup time and keep memory usage to a minimum. All windows and panels you add to your application in Interface Builder will, by default, be deferred.



The Class browser sets the class of the selected window. Unless you've defined subclasses of Window (using the Classes window), this browser displays only Panel and Window. As you would expect, all windows in Interface Builder are instances of the Window class, and all panels are instances of the Panel class.

Given the design of the Window class and the ability of a Window to work closely with a delegate object of your choosing, you'll rarely need to define a subclass. (For more information on delegates, see "Delegates" in Chapter 6.) However, if you do define a new class, this browser lets you establish this class as the class of the selected window.

MenuCell Attributes

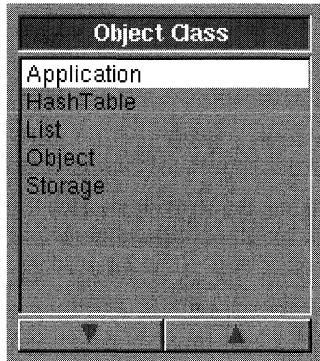


These controls set whether the MenuCell is enabled or disabled and the value of the MenuCell's tag.

An application should disable menu commands that aren't relevant in a particular context. For example, if there's no selection, the Edit menu's Cut command should be disabled. A disabled MenuCell displays gray text and doesn't respond to user actions. If the Disabled button is checked, the selected MenuCell will be disabled at application startup time. An application can dynamically determine which MenuCells are enabled during the course of program execution using the techniques described in "Managing a Window's Display" in Chapter 7.

A MenuCell's tag is an integer that the target of an action message can use to determine which object, out of several capable of sending the same action message, initiated a particular message. By default, all MenuCells have a tag value of 0. (See "Tags" in Chapter 6 for more information on tags.)

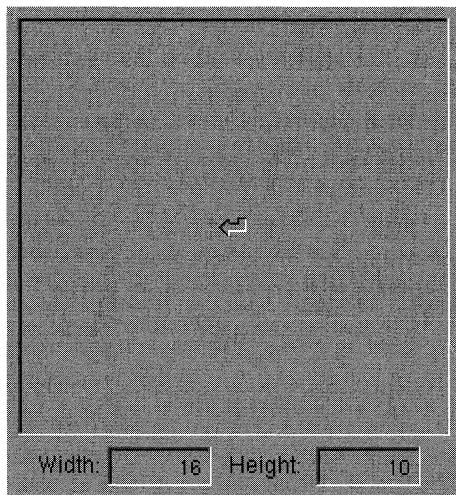
File's Owner's Attributes



This browser sets the class of the interface file's owner object. When you first open this display, the browser displays these classes: Application, HashTable, List, Object, and Storage. If you define additional subclasses of Application or Object using the Classes window, their names will appear here. By selecting a name in the browser and clicking OK, you set the class of the owner object.

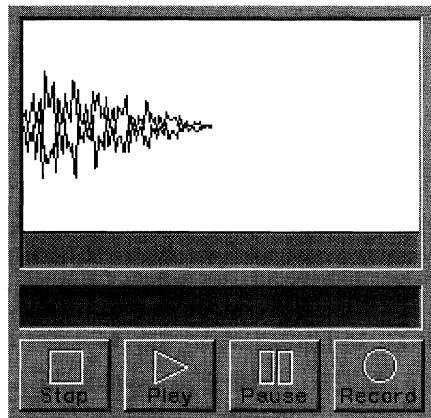
By default, the owner of an application's interface file is an Application object, and the owner object for a module is of the Object class. Depending on your application's design, you can either accept this assignment or change the owner's class to be one of your own definition. (Review the second and fourth projects in the tutorial section at the start of this chapter for examples of both approaches.)

Icon Attributes



Displays the icon that's selected in the Icons window. The Width and Height fields show the dimensions of the displayed bitmap. This display doesn't allow you to edit the image or the dimensions of the bitmap. It's primarily used to examine bitmaps that are too large to be displayed in the Icons window.

Sound Attributes



This display lets you display, record, play, and make limited modifications to sounds.

To show this display, select one of the sound icons in the Sounds window and then open the Inspector window to the Attributes display. The display is divided into three areas. The top portion shows a graphical representation of the sound's waveform. The middle portion displays a horizontal sound meter much like one found on a stereo amplifier. The bottom contains a series of buttons that control the recording and playback of sounds.

Note: The waveform display is provided by an `SoundView` object, and the meter is provided by a `SoundMeter` object. See the sound chapter in the *Sound, Music, and the DSP* manual for more information on these classes.

To play the entire sound depicted by the waveform, click Play. To Play a portion of the sound, select some portion of the sound's waveform and click Play. The standard editing commands Cut, Copy, and Paste operate on the selected portion of the waveform.

The Record button starts recording through a microphone connected to the microphone jack on the back of the MegaPixel Display. If a segment of the waveform is selected when you click Record, the new recording replaces the part of the current recording represented by the selected segment. If a point on the waveform is selected when you click Record, the new recording is inserted at that location in the current recording.

The Pause button halts the recording you're currently creating or playing back. (The bars on the button are highlighted during a pause.) Press Pause again to restart at the point where you paused. The Stop button stops the recording you're currently creating or playing back.

The Connections Display

The Connections display lets you set the outlet (and for Control objects, the action message) for the current connection. Figure 8-35 shows the Connections display for a Form object.

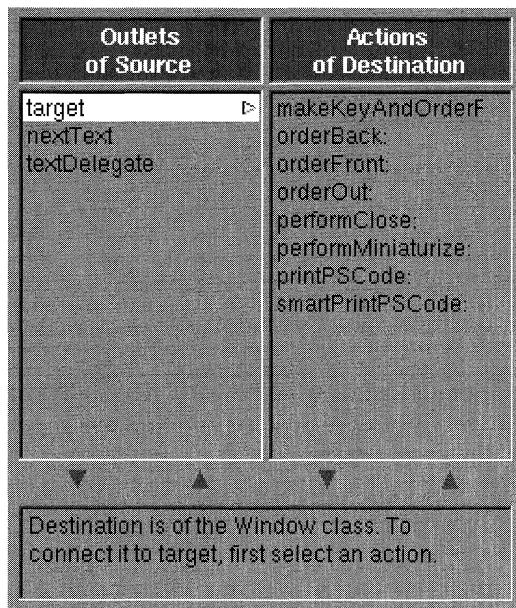


Figure 8-35. The Connections Display

A connection is a communication pathway between a source and a destination object. You create a connection by Control-dragging a line from the source to the destination. When the destination has been unambiguously identified, Interface Builder draws a rectangle around it. Releasing the mouse button completes the operation. If the Inspector window isn't already open, it opens and shows the Connections display.

Note: The rectangle that appears around the destination object is black if the destination object is discrete (such as a single slider or button) and gray if it's a matrix of objects. For example, if you Control-drag a connection toward a Matrix of ButtonCells, a gray rectangle appears around the group of objects when the cursor first intersects the perimeter of the Matrix. As you continue dragging farther into the Matrix the gray rectangle disappears to be replaced by a black rectangle around the ButtonCell that the cursor is within. Thus, Interface Builder lets you connect to the member or the group. Remember that a Form (being a subclass of Matrix) is a matrix even if it has only one FormCell.

The left column of the display lists the outlets of the source. Outlets are instance variables of type **id** that store (by reference) the identity of another object in your application. (See “Outlets” in Chapter 6 for more information.) Application Kit classes have their own predefined outlets; you can add outlets as needed to the classes you create. By selecting an outlet in the left column and clicking the Connect button, you establish an association between that outlet and the destination object. When your application begins running, the outlet variable will be initialized to the **id** of the destination object. Having the **id** of the destination object allows the source object to send it messages.

Control objects have a special outlet, **target**, that identifies the receiver of an action message. (See “Action Messages” in Chapter 7 for details.) If you select **target** as the outlet to be associated with the destination object, the destination object’s action methods are displayed in the right column. By selecting an action message and clicking Connect, you can connect the source Control object with the destination object and specify the action message that the destination will receive.

After you click the Connect button, the button’s title changes to Disconnect, allowing you to remove the connection. If you cut or copy a connected object and then paste it, its connections are severed. To move an object without severing its connections, Alternate-drag it to its new location.

The Autosizing Display

This display lets you specify how an object will respond when its window or superview is resized.

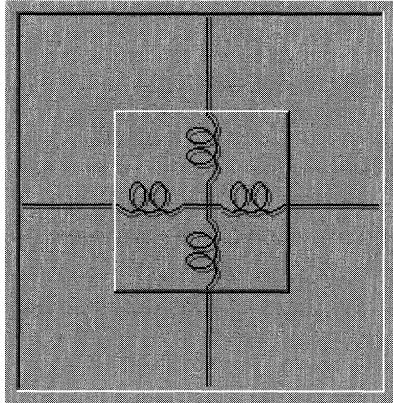


Figure 8-36. The Autosizing Display

The large square represents the window or superview; the small square represents the selected object. The horizontal and vertical lines that bisect the squares are the controls that let you set resizing behavior.

Clicking the horizontal line inside the small square changes the line to a spring shape, indicating that when the window or superview is resized horizontally, the object will also resize to maintain its distance from the left and right margins. In the same way, clicking the vertical line within the object causes the object to become vertically resizable.

The lines outside the object represent the constraints on the object’s distance from the top, bottom, left, and right edges of its superview or window. A straight line indicates that this dimension will remain fixed, if at all possible; a spring shape means that this dimension is resizable. Clicking toggles the image from line to spring shape.

You can create an impossible resizing relationship, such as specifying as fixed the object's dimensions *and* its distance from the window's edges. In cases of conflict, an object's fixed dimension is given precedence over its fixed distance from a border. If all dimensions are made resizable, changes to the window or superview's dimensions are shared by the object and its distance from a border.

The Miscellaneous Display

This display lets you set both the location and the dimensions of the selected View or Window object and its name.

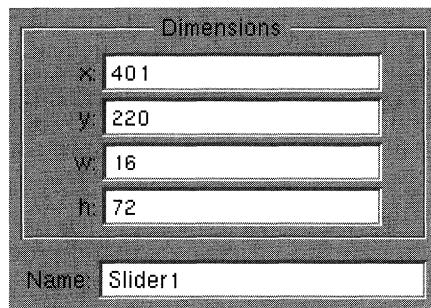


Figure 8-37. The Miscellaneous Display

In Interface Builder, you normally place and resize objects by direct mouse action. This is convenient and efficient; however, if your application requires the precise placement or sizing of objects, you must know their exact coordinates and dimensions. The Miscellaneous display gives you these values.

These fields display the values of the selected object's **frame** instance variable (a structure of type NXRect). For a View, the values are given in the coordinate system of the object's superview. For a Window, the values are given in the screen coordinate system. When you alter any of the values and then click OK, the selected object is relocated and resized to those values. Interface Builder won't let you set an object's **frame** variable to an unreasonable or prohibited value. For example, you can't reset the width of a vertical slider since the Application Kit fixes that value at 16.

The Name field displays the name that Interface Builder has assigned to the selected object. Interface Builder uses these names to keep track of the objects in your interface file (as described in "The File Window" above). In your code, you can also use an object's name to access the object it refers to, although it's generally easier to access an object through an outlet of the interface file's owner.

The Class Display

The Class option in the pop-up list has two effects. It opens the Class Inspector for the class of the selected object. It also brings the Class window forward and highlights the name of the object's class.

The Class Inspector lets you view the outlets and action methods of a class. For some classes (such as Application Kit classes), these class attributes can't be edited; for other classes they can. See the "The Class Inspector" below for more information.

It's important to recognize that the Class option changes the context of the Inspector window from one of editing an object in your application to that of editing the class of an object. You can think of the Class option in an object Inspector window's pop-up list as a convenient link from the inspector for an object to the inspector for the class.

The Project Display

The Project option in the pop-up list opens the Project Inspector. If no project exists in the current directory, the Project Inspector offers to create one. See "The Project Inspector" below for more information.

Like the Class option described above, the Project command changes the context of the Inspector window from one of editing an object or a class to that of editing the project that contains these things. You can think of the Project option as a convenient link between class or object inspectors and the Project Inspector.

The Class Inspector

The Class Inspector lets you view (and, for classes you define, edit) the outlets and actions of a class. The Class Inspector can be opened in either of two ways:

- Select a class in the Classes window and then click the Inspect command in the Windows menu.
- Choose the Class option in the pop-up list at the top of an object or project Inspector window.

In the latter case, before the Class Inspector appears the Classes window comes forward to display the name of the class that will be inspected. The Class Inspector shows the outlets and action methods of that class. For example, Figure 8-38 shows the Class Inspector for the `TextField` class.

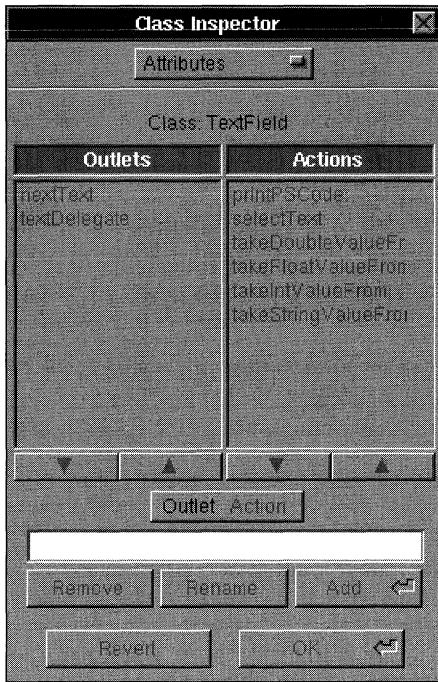


Figure 8-38. The Class Inspector

To edit an object's outlets or actions, click the toggle button below the browser to select the characteristic you want to change. By entering a name in the text field and then clicking the Add button, you can add an outlet or action to the class definition. To remove an outlet or action, select its name in a browser and click the Remove button. To rename an outlet or action, select it in the browser, enter the new name in the text field, and then click Rename. These buttons are disabled if the class you're inspecting can't be edited.

The Project Inspector

As stated in “The Project” at the beginning of this chapter, each application in Interface Builder is part of a project. The project consists minimally of a project directory, a project file (named **IB.proj**) within that directory, and an application’s component files. The project file records such details as the name of the application and of the component files that are required to create it. The Project Inspector lets you edit this information.

From the information stored in the project file, Interface Builder creates a makefile, a file used by the UNIX **make** utility, which oversees the generation of an executable program from source code files. (See the UNIX manual page for **make** for more information.) When you choose the Make command in Interface Builder’s File menu, this utility reads the information in the project’s makefile and builds your application. Thus, the Project Inspector lets you control how your application is built.

You can open the Project Inspector from any of the other inspector displays by dragging to Project in the pop-up list. The Project Inspector has two displays, which are described in the sections that follow.

The Attributes Display

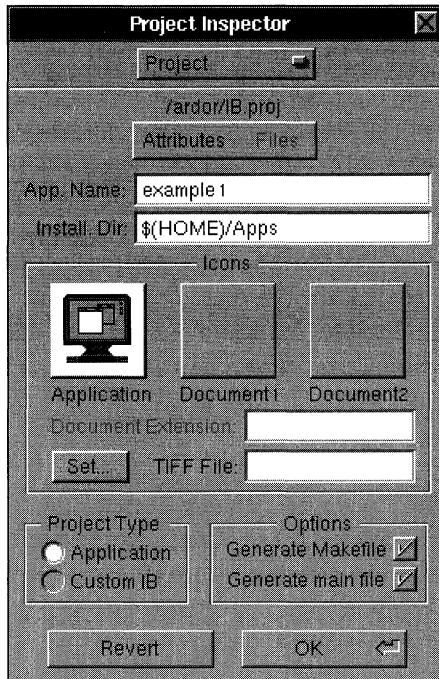
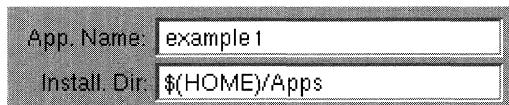


Figure 8-39. The Attributes Display

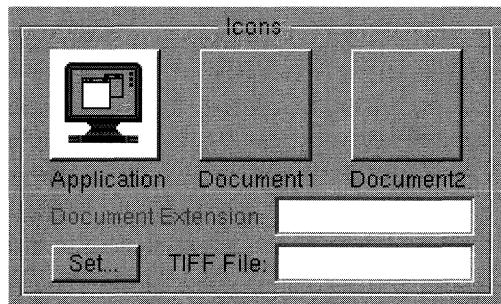


The App. Name field displays the name of the application's executable file. Interface Builder assigns the application the same name as you've chosen for the main interface file—minus the “.nib” extension. The other text field shows the default installation directory, the directory where the finished application's executable file will eventually reside. By default, this is the **Apps** directory within your home directory.

During the building and testing phases of the development process, the application's executable file is kept in the project directory. However, when the application is ready to be released for more general distribution, you can copy the executable file to the installation directory by giving this command in a Shell or Terminal window:

```
make install
```

The **make** utility installs the executable in the directory specified in the Install. Dir. field.



This section lets you set the icons that the Workspace Manager will associate with the application and with its documents.

The top left button displays the application icon; Interface Builder provides you with a generic application icon as shown above. The two buttons to the right are for icons that can be associated with the application's documents, if any.

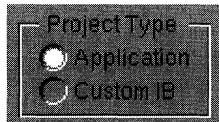
To add an icon to one of the buttons, first click the button to highlight it and then click Set. An open panel appears, letting you specify a file that contains icon data. This data must be in TIFF format and should represent an image no larger than 48 pixels tall and wide. (You can use Icon, an application in **/NextDeveloper/Demos**, to create icons.) You can only change the application icon; it can't be deleted.

Interface Builder keeps track of the icons displayed on these buttons and the files the icons are associated with by writing this information in a file called ***filename.iconheader***, where *filename* is the name of the application. When you compile the application, the header information and the TIFF data for the icons are put in a segment of the application's executable file.

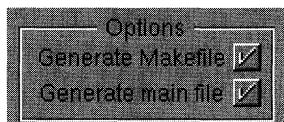
When you create a project, Interface Builder writes this header information to a file named ***filename.iconheader***. Thereafter, whenever the project's makefile is updated, this file containing icon header information is also updated.

Both a document icon and a file extension are required before the Workspace Manager can display an icon for the application's documents. Once the application has been moved to the installation directory (or another directory that the Workspace Manager searches), choose Find Applications in Workspace Manager's Utilities menu to inform the Workspace Manager of the new application and its associated document types.

Be careful not to specify a file extension that's already used by some other application. The Workspace Manager locates the application to open a given document type by following a specific search path. It runs the first application it finds that claims that document type. If, for example, an application in the **Apps** subdirectory of your home directory claims files ending in ".mbox," double-clicking such a file would cause your application, and not Mail, to open the file.



These buttons let you choose the nature of the project, whether you'll be building an independent application or a customized version of Interface Builder. If you're building an entire application or only a module of one, the top button should be checked. If you're customizing Interface Builder (for example, by adding new objects to its Palettes window), the bottom button should be checked.



These buttons let you specify whether Interface Builder will update the project's main program file and makefile. The Generate Makefile button also controls the updating of the icon header file.

When you create a project, Interface Builder generates these three files. Thereafter, whenever you make any changes that would affect these files, Interface Builder will update the files (assuming the appropriate button is checked). For example, adding a source file or a document icon to the project causes the makefile and the icon header file to be updated, and changing the name of the main interface file causes the main file to be updated. In this way, Interface Builder keeps these files consistent with the state of the project.

Although it's generally best to let Interface Builder manage these files, you could edit them directly. If you do, your files could be inadvertently overwritten unless you use these buttons to turn off Interface Builder's automatic update feature.

The Files Display

This display lets you add, remove, and open files in your project.

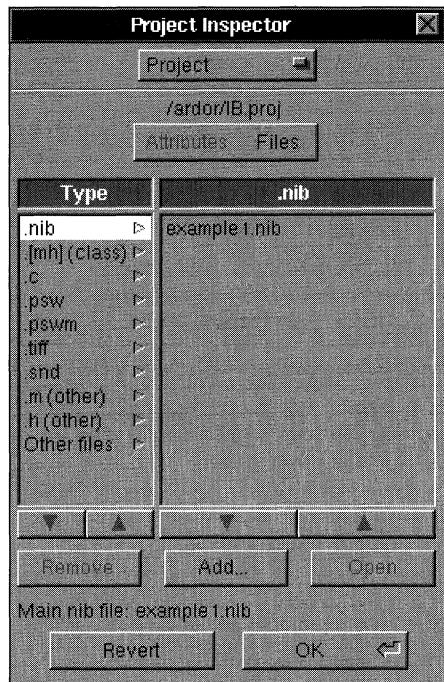


Figure 8-40. The Files Display

The left column lists the types of files that might be part of your project. For a given file type selected in the left column, the right column shows the names of files that are part of the project. The various file types are described below.

File Type	Description
.nib	Interface files. Files created by Interface Builder to hold application interface information.
.m and .h	Files you might create to declare subclasses of Application Kit or other classes. These files contain Objective-C and C source code.
.c	Files containing standard C source code.
.psw	Files that exclusively contain declarations of functions to be created by the pswrap utility.
.pswm	Files that contain declarations for the pswrap utility as well as Objective-C and standard C code.
.tiff	Files that contain TIFF data. In most cases you'll use the Icons window to import TIFF data directly into your executable file. However, this option lets you keep image data in a separate file that's still part of your project.
.snd	Files that contain sound data. This option lets you add large sound files to your project. For sounds of short duration (such as those that you associate with Controls in your application), use the Sounds window to import the data directly into your executable file.
.m (other)	Files containing code that uses Objective-C messaging but that don't declare new classes. Your application's main file will be listed here.
.h (other)	Header files that your application needs beyond those included by the Application Kit or paired with a particular class.
Other files	Any other files you want to be treated as source code for this project.

To add a file to your project, select the file type in the left column and then click Add. An Open panel appears and displays files of this type in the project directory. Since only the files in the project directory can be part of the project, if you attempt to add a file that's not in the project directory, a panel opens asking if you want to copy the file into the project directory.

Selecting a file name in the right column and clicking Open (or simply double-clicking the file name) opens the file for editing.

The Remove button removes the selected file from the project. Removing a file doesn't delete it. It simply removes the reference to the file that was present in the project file, **IB.proj**. The file no longer appears in the Project Inspector, nor is it affected by commands (such as **make install**) that operate on the project's source files.

The text field labeled “Main nib File” displays the name of the project’s main interface file. This field isn’t directly editable: You change its contents by changing the interface files that are part of the project. When you add to the project an interface file that contains a main menu—and no other interface file in the project contain a main menu—the new interface file becomes the main interface file. To replace one main interface file with another, you must remove the old one before adding the new one. No project can have more than one interface file that contains a main menu.

Chapter 9

User-Interface Objects

9-4 The Text Class

- 9-4 Operation
- 9-6 Creating a Text Object
- 9-8 The Text
 - 9-8 Setting the Text
 - 9-9 Examining the Text
 - 9-10 Writing the Text to a File
 - 9-11 Text Layout
 - 9-11 Frame Rectangle
 - 9-12 Margins
 - 9-12 Alignment
 - 9-13 Line and Character Layout
 - 9-14 Word and Character Wrapping
 - 9-15 The Selection
 - 9-16 Setting the Selection
 - 9-17 Querying the Selection
 - 9-18 Text Editing
 - 9-18 Setting Editability
 - 9-18 The Delegate
 - 9-19 Notification Messages
 - 9-19 The `textWillChange:`, `textDidChange:`, and `text:isEmpty:` Messages
 - 9-20 The `textWillEnd:` and `textDidEnd:endChar:` Messages
 - 9-21 The `textWillResize:` and `textDidResize:oldBounds:invalid:` Messages
 - 9-21 The Tag
 - 9-22 Smart Cut and Paste
 - 9-22 Cut, Copy, Paste, and Delete
 - 9-24 The Font
 - 9-24 The Text's Font
 - 9-25 The Selection's Font
 - 9-25 Drawing and the Text Class
 - 9-26 Text Tables and Filter Functions
 - 9-27 The Smart Cut and Paste Tables
 - 9-28 The Click Table
 - 9-28 The Break Table
 - 9-28 Filter Functions
 - 9-29 Character Filter Functions
 - 9-31 Text Filter Functions
 - 9-32 A Text Object in a Scrolling View
 - 9-34 Reusing a Text Object
 - 9-34 Archiving a Text Object
 - 9-35 Rich Text Format Support

9-35 The Box Class

- 9-36 Creating and Modifying a Box Object
- 9-37 The Border
- 9-38 The Title
- 9-39 The Offsets
- 9-40 The Contents of the Box
- 9-40 Box's View Hierarchy
- 9-41 Resizing the Box

Chapter 9

User-Interface Objects

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/ReleaseNotes/AppKit.rtf](#)

The Application Kit consists mainly of class definitions for objects that respond to events or draw on the screen. Chapter 6, “Program Structure,” and Chapter 7, “Program Dynamics,” showed how these objects work together to ground your application in the NeXT window system and give it a unified program structure. Chapter 8, “Interface Builder,” showed how to build a program from these objects graphically, writing only minimal amounts of Objective-C code.

This chapter discusses each of these objects in more detail, from the standpoint of the Application Kit, not Interface Builder. This information will be useful if you want to gain a full understanding of how these objects work, if you plan to define a subclass of any of them to change default behavior, or if you simply want to know the range of possibilities allowed by their kit definitions.

All user-interface objects are Responders, and most are also Views of one sort or another. They fall roughly into five categories:

- The Application object that oversees the entire program. You’d generally customize this object to meet the needs of your program by defining an Application subclass.
- Control objects, Views that implement the target-action paradigm and give users a way to control the program’s activity.
- Window objects that manage the windows provided by the Window Server and the Views that are displayed within them.
- The Text object, which gives programs a way to display text to users and users a way to enter, edit, and select text.
- Views that are generally used in conjunction with other Views, either to provide scrolling and zooming support, or to provide a background and frame to a companion View.

For the most part, these are objects that you can use “off the shelf.” You get an object that’s ready to use just by creating and initializing an instance of a Kit class; you don’t have to define your own subclass.

The Text Class

The Text class is a subclass of View designed for the display and manipulation of text. An object of the Text class can:

- Display text in various fonts.
- Control the format of lines and paragraphs.
- Wrap lines within the boundaries you set.
- Specify displayed text as either read-only or editable.
- Let the user cut, copy, and paste text between windows in the same or different applications.
- Write text to, or retrieve it from, the disk.

A Text object can be used to implement notepads, static text displays, and electronic forms, to name a few possibilities.

Many applications will include a Text object only indirectly, through the use of a TextField or Form object. TextField and Form objects provide a simplified interface to the Text object. They initialize the Text object for you and convert its contents to the format you request: an integer, floating-point, or string value. Since it belongs to a subclass of Control, a TextField or Form object also lets you set an action message and a target. Furthermore, all TextField and Form objects in the same window share a single Text object, thus reducing the memory demands of your application.

TextField and Form give you access to a small but frequently used subset of the methods that the Text class implements. If the text-handling features provided by these Controls don't meet your needs, use a Text object directly. Because the Text class has a rich selection of features, you'll rarely need to create a Text subclass of your own.

Operation

This section gives a general overview of how a Text object works. Keep in mind that through its many methods, a Text object's standard behavior can be altered in fundamental ways.

A Text object is a View specifically designed for text display and editing. It draws in a flipped coordinate system; that is, the origin is located at the bounds rectangle's upper left corner. x-axis values increase from left to right and y-axis values increase from top to bottom. Thus, x values increase for each new character on a line, and y values increase for each new line. See Chapter 6 for general information on View coordinate systems.

Being a View, a Text object renders on the screen only the drawing that lies within its frame rectangle. However, the drawing of characters is confined to a generally smaller area, the *body rectangle*. This rectangle is inset from the edges of the frame rectangle by the width of the Text object's margins, as shown in Figure 9-1. Between the body rectangle and the frame rectangle, a Text object draws only its background color. Within the body rectangle, characters are drawn in horizontal lines that stretch from the left to the right margin. The height of each line is calculated to accommodate the tallest character in the line.

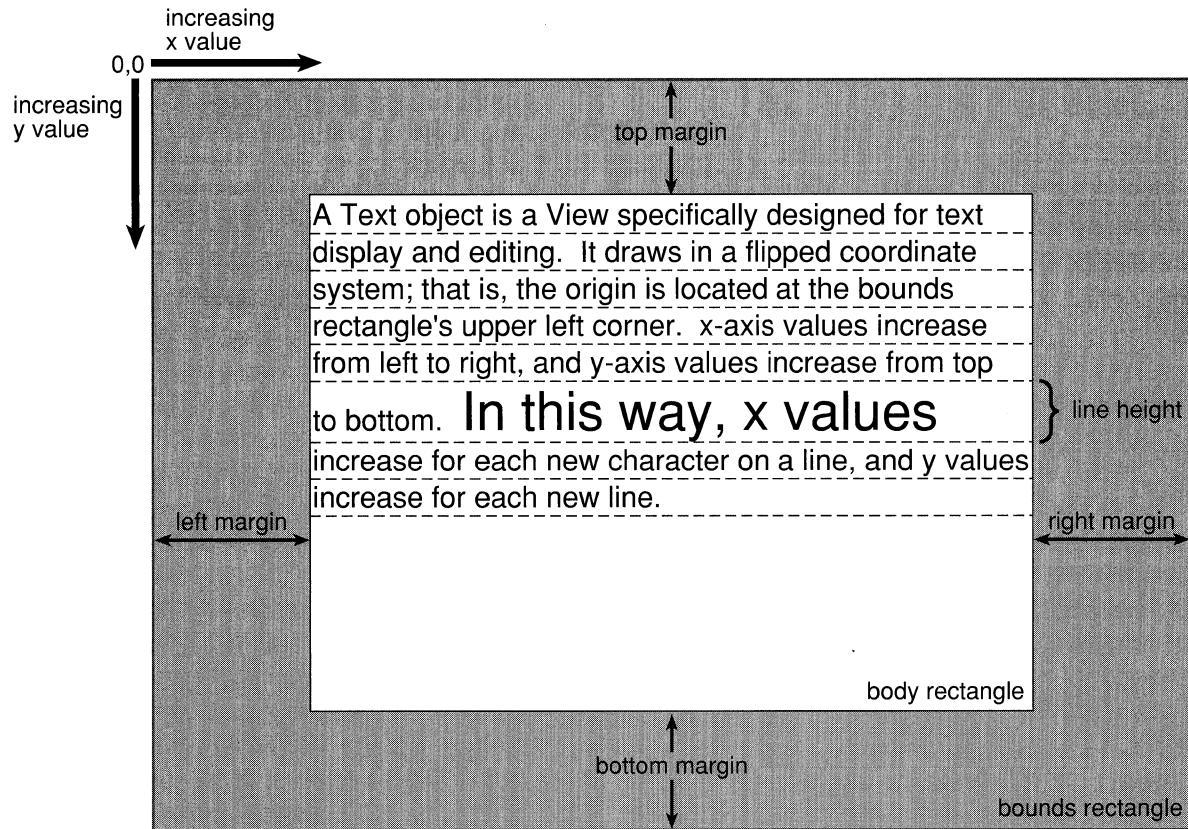


Figure 9-1. Text Layout

A Text object views its contents as a string of characters. Individual characters are identified by their *character position*, their numerical rank in the string starting from 0. For example, if in your application you need to select a group of characters programmatically, you must specify their starting and ending positions.

As a Text object draws a line of text, it continually recalculates the accumulated width of all the characters drawn so far on that line. This calculation takes into account the width of each character in the line (given its font) as well as the width of any tabs or paragraph indentations. If the total width exceeds that available between the left and right margins, a portion of the text is wrapped onto the next line. Lines of text are generally wrapped on a word basis.

By default, the text a Text object displays can be edited by the user. However, there are two mechanisms you can use to protect the text from alteration. One lets you specify that the text is read-only, and the other lets another object within your program decide dynamically when the text can be edited and when it can't. This object is the Text object's delegate.

Besides determining whether the text can be edited, a Text object's delegate can control whether the Text object can resize its frame rectangle in response to the addition or deletion of text and whether it can accept or resign the status of first responder. After allowing any of these changes, the delegate is notified of the Text object's new state.

If its text is editable and the Text object becomes its window's first responder, it receives key events. As key events arrive in the Text object, a *character filter function* examines each character to determine the appropriate action. It passes some characters into the text and interprets others as a signal that the user is finished entering text. By default, Tab and Return characters are accepted into the text, although you can easily configure a Text object to interpret them as a signal to end the editing session.

You can also specify that incoming text, whether from the keyboard, the pasteboard, or a file, is passed to a *text filter function*. This filter is similar to the character filter but more versatile since it's passed more information about the state of the text, the insertion point, and the Text object. By using this added information, a text filter can implement auto-indentation or other automatic formatting features.

The sections that follow describe these concepts in more detail.

Creating a Text Object

You create a Text object by sending a **newFrame:text:alignment:** message to the Text class:

```
myText =[Text newFrame:&aRect text:"Brevity is the soul of wit."
           alignment:NX_LEFTALIGNED];
```

The Text class responds by creating a Text object. The object's frame rectangle, text, and text alignment are set by the method's three arguments:

Argument	Permitted Value
frame	A pointer to an NXRect structure. This structure specifies the frame's size and the location of its lower left corner within the coordinate system of the Text object's superview.
text	A pointer to a null-terminated array of characters. These characters appear when the Text object is displayed.
alignment	A constant specifying the alignment of the text. Text can be aligned with the left or right margin or each line can be centered between the left and right margins. The choices are:

NX_LEFTALIGNED
NX_RIGHTALIGNED
NX_CENTERED

You can also create Text objects by sending **new** or **newFrame:** messages:

```
myText1 = [Text new];
myText2 = [Text newFrame:&aRect];
```

The **new** method sends a message to the Text class to apply the **newFrame:text:alignment:** method. The arguments it sends are a frame rectangle of 0, a null character pointer, and left text alignment. You could then send **sizeTo::** and **setText:** messages to the Text object that's created to set its size and text. (These instance methods are discussed in more detail in the following sections.)

The **newFrame:** method is similar to the **new** method except that it lets you specify the size and location of the Text object's frame rectangle.

The Text object you create will use the default system font. By sending the appropriate message to the Text class, you can specify a different default:

```
[Text setDefaultFont:anObject];
myText = [Text newFrame:&aRect];
```

If you use the **setDefaultFont:** method, be sure that the y-axis of the specified font has the same orientation as that of the Text object. Since the Text object flips the orientation of its y-axis, the font must also have a flipped y-axis to prevent characters from being drawn upside-down:

```
[Text setDefaultFont: [Font newFont:"Times-Roman" size:12.0 style:0
                           matrix:NX_FLIPPEDMATRIX]];
myText = [Text newFrame:&aRect];
```

If you use NULL as the argument to **setDefaultFont:**, the Text object will use the default system font, as defined by the defaults system.

The Text

A Text object provides methods that let you control the form and content of the text. Other methods let you specify whether the user can add, alter, or delete a Text object's text. This section discusses how you can specify what text a Text object displays; the section "Text Editing" discusses the subjects of read-only and editable text.

Setting the Text

The text the Text object displays can be set in a number of ways. It can be:

- Set at the time the object is created
- Set by a message from another object in the application
- Read into the Text object from a file
- Entered by the user

The previous section, "Creating a Text Object," gives an example of setting a Text object's text at the time the object is created. After a Text object has been created, another object in the application can reset the text by sending a **setText:** message:

```
[myText setText:"O, for a horse with wings!"];
```

The argument is a pointer to the array of characters that will appear in the Text object. Any text the Text object previously contained will be replaced by this string. To append text to the contents of a Text object, you have to manipulate the selection, as described in the section "The Selection" below.

You can also set the Text object's text by applying the **readText:** or **readRichText:** method. These methods set the text to the contents of the stream you specify. The stream can represent a Mach port, a portion of memory, or, as in this example, a file.

```
NXStream *input;
char      *theFile = "Sonnets";

input = NXMapFile(theFile, NX_READONLY);
[textOutlet readText:input];
NXCloseMemory(input, NX_FREEBUFFER);
return self;
```

As before, the Text object's previous text is replaced by the new text.

readText: interprets the data in the stream as plain ASCII text. The text is displayed using the Text object's default font and layout characteristics.

A similar method, **readRichText:**, is designed to read text encoded in Rich Text Format[®], or RTF, a format for storing text and graphics. (For more information on RTF, see the *Rich Text Format Specification* by Microsoft Corporation.) The Text object uses the font and layout information embedded in the RTF file as a guideline for displaying the text. If a Text

object doesn't support a specific RTF formatting instruction, it simply ignores it; no error occurs. See the "Rich Text Format Support" section below for a table of supported RTF control words.

Finally, the user can set the text in the Text object. See "Text Editing" below for more information.

Examining the Text

Your application can get a copy of all or part of the Text object's text by sending a **getSubstring:start:length:** message. This method takes three arguments: a pointer to a character array, the starting position of the substring, and the total number of characters to copy into the array. Substring positions are relative to the first character of the text, which is taken to be at position 0.

If you want to copy the Text object's entire text, pick a value for **length** that equals or exceeds the actual text length. No error results if **start** plus **length** specifies a character position beyond the end of the text. For example, to copy the entire contents of **myText**, first use the **textLength** method to determine the text's length. Then, send a **getSubstring:start:length:** message with that value:

```
char *textBuffer;
int textLength;

textBuffer = malloc([text textLength]+1);
[text getSubstring:textBuffer start:0 length:[text textLength]];
```

If the string you request encompasses the entire text, the Text object appends a null terminator ('\0') to the string. As the example illustrates, you should allow for the null terminator when you allocate storage for the string. If you only want a portion of the entire text, you'll have to add the null terminator yourself. This code fragment copies a null-terminated string containing the first 100 characters of **myText**:

```
char textBuffer[101];

[myText getSubstring:textBuffer start:0 length:100];
textBuffer[100] = '\0';
```

getSubstring:start:length: returns an integer indicating the actual number of characters copied. The total doesn't include the null terminator that **getSubstring:start:length:** adds if the requested string includes the entire text. A returned value of -1 indicates that the starting position is beyond the end of the text.

Writing the Text to a File

A Text object can write its text out in either ASCII or RTF format. ASCII format retains only minimal information about the original text. It records the text's characters but no information about fonts, sizes, paragraph indentation, or tab settings. In contrast, Rich Text Format encodes enough information to ensure that the text could be restored to its original specifications. The Text class currently supports only a subset of the formatting commands defined in the RTF specification. See the “Rich Text Format Support” section below for a table of supported RTF control words.

Note: A Text object writes to a stream rather than to a file. (See the “Application Kit Conventions” section of Chapter 6 for information on streams.) The examples that follow assume you want to store the data from the stream into a file.

To write an ASCII version of a Text object’s text to a file, use the **writeText:** method:

```
int      fd;
NXStream *stream;
char    *theFile = "Sonnets";

fd = open(theFile, O_CREAT | O_WRONLY | O_TRUNC, 0666);
stream = NXOpenFile(fd, NX_WRITEONLY);
[myText writeText:stream];
NXFlush(stream);
NXClose(stream);
close(fd);
```

To write only a portion of the text, use **getSubstring:start:length:** to get a copy of the text and then **NXWrite()** to write the string to the stream:

```
int      fd;
NXStream *stream;
char    *theFile = "Snippet";
char    textBuffer[11];

fd = open(theFile, O_CREAT | O_WRONLY | O_TRUNC, 0666);
stream = NXOpenFile(fd, NX_WRITEONLY);
[textOutlet getSubstring:textBuffer start:0 length:10];
textBuffer[10] = '\0';
NXWrite(stream, textBuffer, 10);
NXFlush(stream);
NXClose(stream);
close(fd);
return self;
```

See “The Selection” below for information on writing only the selected text to the disk.

Text Layout

The format of a Text object's displayed text depends on a number of factors, including:

- The size of the Text object's frame rectangle
- The dimensions of the four margins
- The alignment of the text
- The vertical placement of the characters within the line
- The style of each paragraph

Frame Rectangle

As described in “Creating a Text Object” above, you can set the size and location of the frame rectangle when you create a new Text object. A Text object can also dynamically change the size of its frame rectangle in response to the addition or deletion of text. The simplest way to control resizing is with the **setVertResizable:** and **setHorizResizable:** methods. (You can also let the Text object’s delegate control resizing; see “The Delegate” below.) These methods take a boolean value that determines the dimension, width or height, that the rectangle can alter.

```
[myText setVertResizable:YES];  
[myText setHorizResizable:NO];
```

The **isVertResizable** and **isHorizResizable** methods report the resizing status of a Text object.

Assuming the Text object can be resized, the **setMaxSize:** and **setMinSize:** methods set limits to the extent of the change:

```
NXSize maxSize = {100.0, 10000.0};  
NXSize minSize = {100.0, 20.0};  
  
[myText setMaxSize:&maxSize];  
[myText setMinSize:&minSize];
```

This example allows a Text object’s frame rectangle to grow to the specified maximum height as text is added or to shrink to the specified minimum size as text is deleted. Use the **getMaxSize:** and **getMinSize:** method to learn the current settings of these size limits.

If a Text object is resizable, sending a **sizeTo::** message will resize it within the constraints set by the four methods introduced above. Note that **sizeTo::** doesn’t recalculate the placement of text within the newly resized frame rectangle. For this reason, you must send a **calcLine** message immediately after resizing a Text object.

A **sizeToFit** message resizes a Text object so that all of the text it contains is displayed. Again, this resizing is subject to the limits set by the four methods above. Use **calcLine** to rewrap the text after sending a **sizeToFit** message.

Margins

Text layout is also determined by the width of a Text object's margins. The margins determine the amount the body rectangle is inset from the sides of a Text object's frame rectangle. By default, the body rectangle's height is the same as that of the frame rectangle, but its width is 4.0 units (in the Text object's current coordinate system) narrower than the width of the frame rectangle. The body rectangle is centered between the left and right edges of the frame rectangle.

The **getMarginLeft:right:top:bottom:** method reports the current settings of the four margins. To alter these values, send a **setMarginLeft:right:top:bottom:** message. With this method, you can reset all four values at once. You might use both methods if you wanted to reset only one margin. For example, to reset the top margin to 6.0 while leaving the other margins at their current settings, you could send these messages:

```
NXCoord lMgn, rMgn, tMgn, bMgn;  
  
[myText getMarginLeft:&lMgn right:&rMgn top:&tMgn bottom:&bMgn];  
[myText setMarginLeft:lMgn right:rMgn top:6.0 bottom:bMgn];
```

Alignment

Text alignment also affects the layout. By default, text is aligned with the left margin, which causes a block of text to have a “ragged right” edge. After a Text object is created, its text alignment can be queried by sending an **alignment** message and can be reset by sending a **setAlignment:** message.

The **setAlignment:** method uses the same constants (NX_LEFTALIGNED, NX_RIGHTALIGNED, and NX_CENTERED) as the **newFrame:text:alignment:** class method described in “Creating a Text Object” above. As the names of the constants suggest, text can be aligned to the left or right margin or centered between the left and right margins.

Sending a **setAlignment:** message doesn't redraw the text; you must redisplay the text to exhibit the change. For example:

```
if ([myText alignment] != NX_CENTERED) {  
    [myText setAlignment:NX_CENTERED];  
    [myText display];  
}
```

Line and Character Layout

Methods defined in the Text class let you adjust the vertical placement of characters within a line as well as the height of the line itself, giving you the ability to display single-spaced or double-spaced text, or any other spacing you choose. The default values for line height and character placement depend on the Text object's font, but are designed to give single-spaced text for the font that's currently in use.

In the context of the Text class, a line is a rectangular area that extends from the left to the right edge of the Text object's body rectangle and that contains a single row of characters. Each line within a paragraph shares a side with the preceding and the following line; thus setting the line spacing for a Text object is, in fact, a matter of setting the height of the lines.

Within a line, a Text object positions a character according to two measurements. It sums the widths of the preceding characters to determine the x-coordinate, and it sets the y-coordinate a certain distance above the bottom of the line's rectangular drawing area. Figure 9-2 illustrates character placement in a line of text.

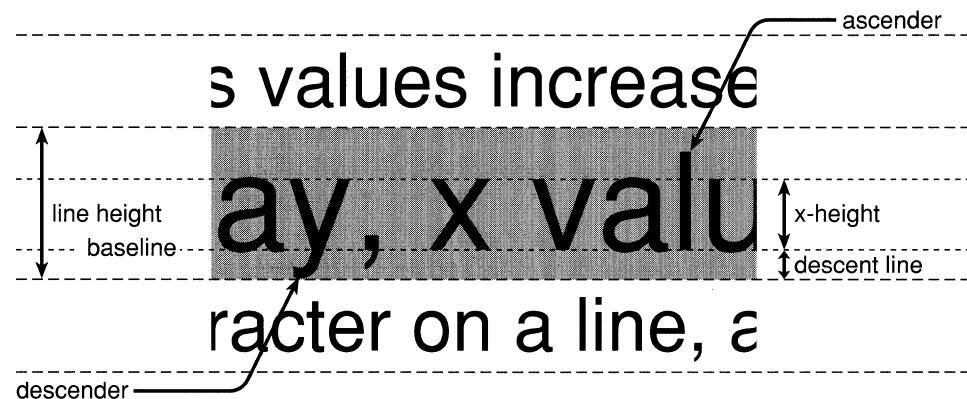


Figure 9-2. Line and Character Metrics

Characters within a font are horizontally aligned relative to a reference line called the *baseline*. You can think of characters such as “x” and “h” as standing on the baseline. For lowercase letters, the vertical extent of a letter is divided into three parts: the ascender, the descender, and a middle section. The middle section extends from the baseline to the top of an “x” character, a height known as the *x-height*. The portion of a character that extends above the x-height is called the *ascender*; the portion below is called the *descender*. The distance from the baseline to the bottom of the line's drawing area is called the *descent line*.

The Text class's **setLineHeight:** and **lineHeight** methods set and report a Text object's line height. To display double-spaced lines of text, you would reset the Text object's line height. Assuming the Text object's text is currently single-spaced, you could specify double-spaced text by sending these messages:

```
[myText setLineHeight:2.0 * [myText lineHeight]];
```

To adjust the vertical placement of characters within a line, you can change the descent line. The Text object's **descentLine** and **setDescentLine:** report and set the descent line. For example, to double the current value of the descent line:

```
[myText setDescentLine:2.0 * [myText descentLine]];
```

Although you might specify a line height or descent line value that would cause a character to extend outside of a line's rectangular drawing area, a Text object will override these values so that even the largest character on the line is fully visible.

Word and Character Wrapping

A Text object continues to draw characters on a single line until adding another character would cause the total width of the characters to exceed the space between the line's right and left margins. What happens next depends on the previous characters on the line and the current state of the Text object:

- The new character, and the word it's part of, can appear on the next line: The text is *word-wrapped*.
- The new character alone can appear on the next line: The text is *character-wrapped*.
- The new character can be added to the Text object's text but not be drawn within the frame rectangle: The drawing of the new character, and any that follow it on the same line, is clipped by the Text object's body rectangle.
- The width of the Text object's frame rectangle can grow to accommodate the new character.

A Text object's default behavior is to word-wrap lines of text. If adding text to a line would push some character on the line outside the body rectangle, the Text object takes the word this character is part of and redraws the entire word on the next line. Unlike a UNIX text editor such as Emacs, a Text object wraps the text without adding a new line character ('\n') at the end of each line. Wrapping affects only the display of the text, not its contents. Consequently, if you change the width of a Text object's body rectangle, the text it contains will be rewrapped to the new line length.

To determine word boundaries, a Text object refers to a *break table*. Although it's unlikely you'll ever need to change the standard break table, the Text class provides the **setBreakTable:** and **breakTable** methods to give you that option. See "Text Tables and Filter Functions" below for more information on the break table.

If the Text object can't find a word boundary on the current line (that is, a single "word" spans the body rectangle's width), it wraps new characters onto the next line. The **setCharWrap:** and **charWrap** methods let you control how a Text object treats such long words:

```
if ([myText charWrap] == YES)
    [myText setCharWrap:NO];
```

After receiving a **setCharWrap:NO** message, a Text object's word wrap feature is still enabled, but the display of long words is clipped to the edge of the body rectangle. Although the clipped characters aren't visible, they're still part of the Text object's text. By deleting characters that precede them on the line, or by entering Return in the midst of the long word, the user can bring the clipped characters into view.

You can disable both word-wrapping and character-wrapping by sending a **setNoWrap** message:

```
[myText setNoWrap];
```

After a Text object receives a **setNoWrap** message, a user can begin a new line only by entering Return. A Text object allows no more than 16,384 (or 2^{14}) characters on a single line.

If the Text object is resizable, it can grow to accommodate text that would exceed the width of the body rectangle. If the enlarged frame rectangle still falls within its superview's frame rectangle, the new text will be visible. See the "Frame Rectangle" section above and "The Delegate," below, for more information.

The Selection

The selection is the portion of the text that an action, such as a command, will affect. The selection can have 0 width (an *empty selection*), in which case it's indicated by a blinking vertical bar called the *caret*, or it can contain one or more characters. If the selection spans one or more characters, it's indicated by highlighting. A Text object automatically chooses the proper marking depending on the length of the selection.

Highlighting is only an attribute of the selection; it shouldn't be confused with the selection itself. By setting the selection's gray value to be equal to that of the background (see "Drawing and the Text Class" below), you can easily create a selection that doesn't display a visible highlight.

By default, a Text object's text is selectable. The **isSelectable** method reports whether the text is selectable, and the **setSelectable:** method lets you alter this status:

```
if ([myText isSelectable] == YES)
    [myText setSelectable:NO];
```

Only selectable text is editable; however, selectability doesn't necessarily imply editability. You can have selectable, read-only text. For example, you may want the user to select some text, say the answer to a multiple-choice question, but not be able to alter that text. The "Text Editing" section that follows describes how to protect text from alteration.

The Text class provides methods that affect a selection's length, character contents, font, gray value, and location in a scrolling view. A selection's font and gray value are discussed later under "The Font" and "Drawing and the Text Class"; the other concepts are introduced in the next two sections.

Setting the Selection

The selection can be set by the user or it can be set programmatically. In most cases, the selection is set by the user's actions with the mouse, by clicking, dragging, or a combination of the two. See Chapter 2, "The NeXT User Interface," for a general discussion of the semantics of mouse actions. The rest of this section discusses how to manage the selection programmatically.

The visibility of the caret, which marks the empty selection, is generally determined by the Application Kit itself rather than by the code you write. When a Text object becomes the first responder, it sends itself a **showCaret** message to start the timed entry that blinks the caret. Conversely, before a Text object stops being the first responder, it sends itself a **hideCaret** message to erase the caret from the text and to remove the timed entry. You'll rarely need to send these messages in your own code.

The position and extent of the selection can be set programmatically using the **setSel::** method. This method takes two integer arguments corresponding to the first and last character positions of the selection. If the two arguments are identical, **setSel::** has the effect of setting the position of the caret.

```
[myText setSel:0:0];
```

In this case, the caret moves to the start of the text. Note that the selection is measured from the left edge of the character at the specified position. Character position 0 is the position of the first character of the text; sending a **setSel:0 :0** message moves the caret immediately to the left of the first character.

To select the first ten characters of the text, send this message:

```
[myText setSel:0 :10];
```

Since the selection is measured from the left edge of the character position, this message selects the characters at character position 0 through 9.

The Text class provides the **selectText:**, **selectAll:**, and **selectError** methods for selecting the entire text. **selectText:** and **selectAll:** are synonyms. They attempt to make the text object the first responder and then, if successful, select the entire text. The **selectError** method is similar to these other two except that it doesn't attempt to alter the Text object's

responder status before selecting the text. These methods are used primarily in support of the Text class's delegate/notification system, as described in "The Delegate" below.

Querying the Selection

The **getSel::** method returns the starting and ending positions of the current selection, along with other information. Instead of taking **int** arguments like **setSel::**, **getSel::** takes pointers to **NXSelPt** structures. An **NXSelPt** structure has these fields:

```
typedef struct _NXSelPt {
    int      cp;      /* the character position */
    int      line;    /* line offset */
    NXCoord x;       /* x coordinate of character position */
    NXCoord y;       /* y coordinate of character position */
    int      c1st;   /* character position of first character in line */
    NXCoord ht;     /* line height */
} NXSelPt;
```

The first field, **cp**, gives the character position of one end of the selection. The value of **line**, interpreted in light of information that a Text object keeps in an internal table of line breaks (see "The Break Table" in "Text Tables and Filter Functions" below), identifies the line of text containing this character position. The next two fields give the x and y location of the top corner of this end of the selection. **c1st** identifies the character position of the first character of the line containing **cp**. The last field, **ht**, gives the height of the line containing **cp**. Two **NXSelPt** structures give a Text object the information it needs to identify and highlight the selected characters.

Using the character position information from the **NXSelPt** structure, you could copy the selected text to another buffer:

```
NXSelPt  selStart, selEnd;
char      buf[100];
int       selLength;

[myText getSel:&selStart :&selEnd];
selLength = selEnd.cp - selStart.cp;
[myText getSubstring:buf start:selStart.cp length:selLength];
buf[selLength+1] = '\0';
```

The **replaceSel:** method lets you alter the contents of the selection. With this method, for example, your application could replace a selected misspelled word with the correct spelling. You can also use **replaceSel:** to append text to the current contents of a Text object:

```
int      length;

length = [myText textLength];
[myText setSel:length :length];
[myText replaceSel:"\n\nYours truly,\n"];
```

Text Editing

The Text class provides two ways to control the editability of a Text object's text; the approach you take will depend on your needs. To set the editing status for the duration of the program's execution, use the **setEditable:** method. To allow editing under some circumstances but not others, let the Text object's delegate determine if and when the text is editable.

Note that these two systems can't be used at the same time. Sending a **setEditable:** message disables the delegate/notification system.

Setting Editability

The **setEditable:** method lets you specify the editability of the text, and the **isEditable** method reports the text's current editing status:

```
if([myText isEditable] == YES);  
[myText setEditable:NO];
```

The following sections describe how a delegate can control the editability of the text.

The Delegate

A Text object can send notification messages to a delegate in response to user actions. The delegate's response determines the Text object's behavior, such as whether it will allow the user to edit its text.

You set and query a Text object's delegate just as you would a Window's:

```
[myText setDelegate:anObject];  
anObject = [myText delegate];
```

A Text object sends predefined messages to its delegate depending on the user's specific input. These messages are described in the next section.

When you set the delegate, the Text object queries it to discover which of the predefined messages it responds to. Thereafter, the Text object only sends those messages that the delegate can accommodate. As noted below, a Text object reverts to its default behavior if its delegate doesn't respond to a particular message, or if there is no delegate object.

If, in your application, more than one Text object reports to the same delegate, you'll need a way to identify which text object sent the notification message. See "The Tag" below for such an identification system.

Notification Messages

The notification messages a Text object sends to its delegate can be divided into two categories: those that are sent before a change is made and those that are sent after. Messages in the first group advise the delegate of an impending change, giving the delegate the chance to allow or prevent the change. Messages in the second group simply report that a change has been made.

Before Change	After Change
textWillChange:	textDidChange:
	text:isEmpty:
textWillEnd:	textDidEnd:endChar:
textWillResize:	textDidResize:oldBounds:invalid:

The textWillChange:, textDidChange:, and text:isEmpty: Messages

The delegate receives a **textWillChange:** message the first time the user attempts to change the text after a Text object has become the first responder. Only one **textWillChange:** message, the one accompanying the first attempted change, is sent during the time the Text object retains first responder status. The delegate, depending on the value returned by its **textWillChange:** method, either allows or prevents the change.

If the delegate's **textWillChange:** method returns NO, the change is allowed; if it returns YES, the change is prevented. You can think of a NO return value as indicating that the delegate has "no objection" to changes in the Text object.

```
- (BOOL) textWillChange: (id) textObject
{
    return (NO);
}
```

If the delegate doesn't implement the **textWillChange:** method, or if the Text object has no delegate, the change is allowed by default.

If the delegate's response to the **textWillChange:** message allows the change, the delegate immediately receives a **textDidChange:** message. Thereafter, each time the user changes the text either by using the keyboard or the pasteboard, the delegate receives a **text:isEmpty:** message. This message passes the Text object's **id** and a boolean value indicating whether the Text object contains any text after the change. One example of the use of this information is the Save panel. The OK button is enabled or disabled depending on whether there's text in the text field provided for the file name.

When a user is entering characters from the keyboard, the Text object attempts to send a **text:isEmpty:** message for each character. A fast typist might outrun this system momentarily, but it's unlikely such a typist could enter more than two or three characters before the Text object sends a message. When a user cuts text from or pastes text into a Text object from the pasteboard, the Text object sends a single **text:isEmpty:** message to alert the delegate of the change.

*The **textWillEnd:** and **textDidEnd:endChar:** Messages*

Before a Text object ceases to be the first responder, it sends a **textWillEnd:** message to its delegate. The delegate can either allow the change or prevent it, depending on the return value of its **textWillEnd:** method.

If the delegate objects to the change, its **textWillEnd:** method returns YES; otherwise, it returns NO. If the delegate objects, the Text object remains the Window's first responder and selects the entire text (see “The Selection” below).

The **textWillEnd:** message gives the delegate a chance to validate the Text object’s text before letting the user select some other object:

```
- (BOOL) textWillEnd:(id)textObject
{
    char *modelBuffer = "The accepted answer.";
    char *textBuffer;

    /* get entire text */
    textBuffer = malloc([text textLength]+1);
    [myText getSubstring:textBuffer start:0
                  length:[myText textLength]+1];

    /* compare the text to the model */
    if (!strcmp(textBuffer, modelBuffer))
        /* they're the same */
        return(NO);
    else
        /* they're different */
        return(YES);
}
```

If the delegate doesn’t implement the **textWillEnd:** method, or if the Text object has no delegate, the change is allowed by default.

After a Text object ceases to be the first responder, it sends a **textDidEnd:endChar:** message to its delegate. The two arguments passed with this message are the Text object’s **id** and the character that caused the Text object to stop being the first responder. The delegate can use this information to decide which other object should become the first responder. For example, if the character was Tab, the delegate would typically change the first responder to the next Text object in the window.

The Text object’s character filter determines which characters the Text object interprets as a command to cease being the first responder. See “Character Filter Functions” below for more information.

The textWillResize: and textDidResize:oldBounds:invalid: Messages

The Text object's bounds rectangle can change size to accommodate a change in the amount of text. The bounds rectangle can either grow to hold an increased amount of text or shrink if some of the text is deleted. In either case, before the change can take place, the Text object sends a **textWillResize:** message to its delegate, passing its **id** as the message's argument.

As with the **textWillChange:** and **textWillEnd:** methods, the delegate's **textWillResize:** method can prevent the bounds rectangle from changing size by returning YES. If, on the other hand, the delegate permits the Text object's bounds to change size, it can specify to what degree and in which directions the change can occur:

```
- (BOOL) textWillResize:(id)textObject
{
    NXSize maxSize = {100.0, 10000.0};
    NXSize minSize = {100.0, 100.0};

    [myText setMaxSize:&maxSize];
    [myText setMinSize:&minSize];
    return NO;
}
```

This **textWillResize:** method lets the Text object's bounds rectangle grow to 10000.0 units in the Text object's coordinate system in the vertical (or y) direction. No growth is allowed in the horizontal (or x) direction. It's not permitted to change size in either direction unless the Text object has been configured to be resizable as described in "Frame Rectangle" above.

If the delegate doesn't implement this method, or if the Text object has no delegate, the change is allowed by default.

The Text object sends a **textDidResize:oldBounds:invalid:** message to its delegate after the bounds rectangle's size has been changed. The Text object passes its **id** as the first argument and its old bounds rectangle as the second argument. The third argument is the area of the Text object's superview that should be redrawn if the Text object's new bounds rectangle is smaller than its old bounds rectangle.

The Tag

You can assign a tag, an arbitrary integer, to a Text object as a means of identifying it to its delegate:

```
[myText1 setTag:1];
[myText2 setTag:2];
```

A tag gives the delegate a way to determine which Text object, among several, has sent a particular notification message:

```
- (BOOL) textWillChange:(id)textObject
{
    if([textObject getTag] == 1)
        /* prevent the change */
        return(YES);
    else if([textObject getTag] == 2)
        /* allow the change */
        return(NO);
}
```

Smart Cut and Paste

A Text object attempts to respect word spacing conventions when the user cuts words from or pastes them into the text. For example, consider this sentence:

“I see no objection to stoutness, in moderation.”
—W.S. Gilbert 1836-1911

If the user double-clicks the word “no” and then cuts it, the word and space character to its left disappear. This leaves only one space between the words that had flanked “no”. Furthermore, if the user double-clicks the word “moderation” and then cuts it, the period is brought up tight to the word “in”. Similarly, a word pasted into a line of text adds a space character when needed. For example, pasting the word “strong” between the words “no objection” results in “no strong objection”. It doesn’t matter if the insertion point is to the left or right of the intervening space, standard word spacing is preserved. Pasting a word between an existing word and its following punctuation likewise leaves no unwanted spaces. Note that a Text object only uses “smart” cut and paste on word selections. The user must double-click, double-click and then Shift-click to extend the word selection, or double-click and drag to select a series of words to invoke this feature of the Text class.

See “The Smart Cut and Paste Tables” below for details on how this feature is implemented.

Cut, Copy, Paste, and Delete

The Cut, Copy, Paste, and Delete commands of a standard application’s Edit menu send, respectively, **cut:**, **copy:**, **paste:**, and **delete:** messages to the first responder. If you create an application using Interface Builder, these messages are already assigned to the appropriate menu item. A Text object is typically a Window’s first responder and so must interpret these messages as editing commands.

Note, however, that not all Text objects may become first responder. To become first responder, a Text object must be selectable and, if it has a delegate, the delegate must allow

the change of status. If a Text object doesn't meet these criteria, these editing commands will have no effect on it.

The simplest of these four editing methods is **delete:**. Assuming the current selection includes one or more characters, a **delete:** message removes the selected text from the Text object. The Text object rewraps and redisplays the remaining text and replaces the extended selection with the caret. If there is a delegate, and it implements the method, the Text object sends it a **text:isEmpty:** message. A **delete:** message affects only the text: No text is placed on the pasteboard.

copy: is the complement of **delete:**. A **copy:** message affects only the contents of the pasteboard, leaving the text unaltered. Again assuming a selection of positive length, a **copy:** message opens the pasteboard and writes the selected text to it. The Text object puts three types of data relating to the selection on the pasteboard:

- The text
- Format information
- Selection-type information

The text is simply an ASCII string of a specified length. Format information consists of the font characteristics and paragraph styles included in the copied text. The selection type records whether the text was the result of word selection. A Text object that receives this text can use the selection-type information to determine correct word spacing for the inserted text. Since a **copy:** message doesn't alter the text, no notification message is sent to the Text object's delegate.

The **cut:** method combines the functions of the **copy:** and **delete:** methods. In fact, when a Text object receives a **cut:** message, it sends itself a **copy:** message to copy the selected text to the pasteboard and then sends itself a **delete:** message to remove the selected text.

Assuming a Text object contains editable text, a **paste:** message causes it to read the contents of the pasteboard and insert the text. If the current selection is marked by a caret, the text is inserted at that point. If the current selection includes one or more characters, the inserted text replaces these characters. In either case, the Text object places the new insertion point after the inserted text.

Before inserting the pasteboard's contents into the text, a Text object checks whether this text was the result of word selection. If so, the Text object refers to its smart cut and paste tables to determine whether a space character should be added to one or both ends of the string to maintain proper word spacing around the inserted text.

The pasteboard can also contain formatting information for the text it contains. Depending on the Text object's state, it can either preserve the inserted text's previous format or ignore it and make the new text conform to the Text object's prevailing format. By default, a Text object ignores the formatting information from the pasteboard. The **setMonoFont:** method, as described in "The Text's Font" below, controls how a Text object handles formatting information from the pasteboard.

If the Text object's delegate implements the corresponding method, it receives a **text:isEmpty** message when text is pasted into the Text object.

The Font

The Text class defines methods for setting the name, size, and style of the font for the entire text or any portion of it. A Text object alters the characteristics of a font by sending messages to a Font object. Although you can set a Text object's Font object directly, it's generally easier to use Text methods that let you set the individual font characteristics. The two sections below introduce the methods that affect a Text object's fonts.

A Text object actually receives two fonts for each font it requests. These two fonts are identical except for character width information: One font is designed for screen use and the other for the printer. In determining line breaks for some text, a Text object makes two calculations, one using screen widths and the other using printer widths. The Text object compares the results of these calculation and then breaks each line based on the result of the calculation that leaves the least text on a line. In this way, when comparing printed output to text on the screen, line breaks are consistent although precise line lengths may differ.

The Text's Font

You can set the default font for a Text object using **setDefaultFont:**, as discussed in “Creating a Text Object” above. Once a Text object is created, use the **setFont:** method to set the font of the entire text. This method takes a Font **id** as its sole argument:

```
[myText setFont: [Font newFont:"Helvetica" size:12.0 style:0  
matrix:NX_FLIPPEDMATRIX]];
```

Notice that because a Text object draws in a flipped coordinate system, the matrix of the font it uses must match this orientation.

Since altering the font may change the number of characters that fit on a line, after receiving a **setFont:** message, the Text object recalculates the line layout and redraws the text.

When text is cut or copied to the pasteboard, it retains its font characteristics. When this text is pasted into another Text object, the pasted text can be displayed with its original font characteristics or it can assume those of the destination Text object. For many uses, it's more convenient to have the Text object use a single font. For example, a programmer writing a data-analysis application generally wouldn't want the added complexity of multiple font characteristics.

A **setMonoFont:** YES message makes a text object ignore the font information that accompanies pasted text. This is the default state of a Text object. The pasted text is displayed using the font characteristics of the first character of the existing text. The **isMonoFont** method reports whether a Text object will ignore the font information of pasted text.

Methods that change the font of selected text or read in text containing multiple fonts shouldn't be used with a Text object that is configured to contain only a single font. A text object configured in this way can display multiple fonts; however, each time text is cut or copied and then pasted, font information will be lost.

The Selection's Font

The Text class defines a set of methods that let you alter the selection's font:

```
setSelFont:  
setSelFont:paraStyle:  
setSelFontFamily:  
setSelFontSize:  
setSelFontStyle:
```

setSelFont: takes a Font object as an argument, allowing you to set several font characteristics simultaneously. The related method, **setSelFont:paraStyle:**, lets you additionally set the paragraph style of the selected text.

The next three methods each affect only one characteristic of the selection's font. **setSelFontFamily:** takes a single argument specifying a font name, such as Helvetica or Times-Roman. **setSelFontSize:** and **setSelFontStyle:** set the size and style of the selection's font.

As with **setFont:**, after a Text object resets the selection's font in response to any of the above messages, it recalculates the line breaks and redraws the text.

Drawing and the Text Class

A Text object, by default, is a transparent View: Sending it a **display** message doesn't cause it to paint every pixel within its frame rectangle. (View transparency is discussed under "The Display Methods" in Chapter 7.) If a Text object's text is set in some way other than the keyboard (for example, by using a method such as **readText:** or **setText:**), the Text object draws only the characters on each line of text, not the pixels surrounding them. This allows you to draw text over another View without affecting more of that View than is necessary to display the characters.

However, when a Text object is receiving input from the keyboard, it always redraws the complete line—characters and background—that the new characters fall on. If, after the user is through entering text, you want to restore the background, send the Text object a **displayFromOpaqueAncestor:::** message.

To force a Text object to repaint all the area within its frame rectangle whenever it receives a message to display itself, send it a **setOpaque:** message:

```
[myText setOpaque:YES];
```

A Text object, by default, draws black characters on a white background and marks the selection with by a light-gray highlight. You can change the gray value of the background, text, or selection by sending the appropriate messages.

The gray values of the background, text, and selection can be set to any value from 0.0 (represented by the constant NX_BLACK) to 1.0 (represented by the constant NX_WHITE) as the gray value argument of these methods. However, since only NX_WHITE, NX_LTGRAY, NX_DKGRAY, and NX_BLACK produce pure values on the NeXT computer's screen, these values result in the most legible text displays.

To invert the gray values of the standard text display, you would send these messages:

```
[myText setBackgroundGray:NX_BLACK];
[myText setTextGray:NX_WHITE];
[myText setSelGray:NX_DKGRAY];
```

The **backgroundGray**, **textGray**, and **selGray** methods return the current values for these three parameters.

The caret is always black, so setting the background gray value to black will obscure the caret.

Text Tables and Filter Functions

Much of a Text object's functionality is determined by the data it finds in certain tables and by the paradigms embodied in certain functions. The Text class provides methods that set these tables and functions, allowing you to change a Text object's behavior in fundamental ways without altering its class interface. This section introduces some of these tables and functions.

The Smart Cut and Paste Tables

As described in “Smart Cut and Paste” above, a Text object tries to maintain, during cut and paste operations, the proper relationship between words and their surrounding text. It does this by consulting the two tables that are referred to by its **preSelSmartTable** and the **postSelSmartTable** instance variables. The Text class provides the **setPreSelSmartTable:** and **preSelSmartTable** methods to set and return the former table and a matching pair of methods to manage the latter table. These smart cut and paste tables define which characters a Text object should consider equivalent to a space.

For smart cut and paste to work, the selection must be the result of word selection. When you select a word or group of words and then cut the selection, the Text object removes the selection and the space to its left (if any). However, only the text is stored on the pasteboard; the space character is ignored. The Text object also records on the pasteboard that the cut text resulted from a smart cut operation.

When you paste the pasteboard’s contents into the text, the Text object first checks to determine whether the pasteboard contains text that is the result of a smart cut or copy operation. If it does, the Text object looks at the characters that border the right and left sides of the insertion point or selection to be replaced. If the character on the left is in its smart left paste table, it pastes the word in without adding or subtracting anything on that side. If the character isn’t in the left table, the Text object adds a space on that side when it pastes the word. It uses the same process on the right.

Here are the tables that the Text object uses by default:

```
unsigned char NXSmartLeft[] = {' ', NX_EMSPACE, NX_ENSPACE,
    NX_THINSPACE, NX_FIGSPACE, '\n', '(', '\t', '[', '\320', '\'', '\'', '\0'};

unsigned char NXSmartRight[] = {' ', NX_EMSPACE, NX_ENSPACE,
    NX_THINSPACE, NX_FIGSPACE, '\n', ')', '\t', ']', '.', ',', ';',
    ':', '?', '\'', '!', '\'', '\0';
```

So, for example, pasting a word immediately to the right of a Tab character won’t cause a Text object to add a space character on that side. Similarly, pasting a word immediately to the left of a colon doesn’t cause a Text object to add a space between the word and the colon.

If you are going to use a Text object as an editor for C language programs, for example, you might prefer both tables to look like this:

```
unsigned char smartForC[] = {' ', '\200', '\201', '\202', '\203',
    '\n', '\t', '!', '\'', '#', '$', '%', '&', '^', '(', ')', '*',
    '+', ',', '-.', '.', '/', ':', ';', '<', '=' , '>', '?', '@', '[',
    '\', ']', '^', '_', '}', '{', '|', '}', '~', '\320'}
```

To set this table for both the **preSelSmartTable** and the **postSelSmartTable**, you’d send these messages:

```
[text setPreSelSmartTable:smartForC];
[text setPostSelSmartTable:smartForC];
```

This table illustrates the different effects of the default tables in comparison to the C language table:

	Before Cut	After Cut	After Paste
Default	!word!	!!	! word!
C Table	!word!	!!	!word!
Default	(word((((word (
C Table	(word((((word(
Default	*word*	**	* word *
C Table	*word*	**	*word*

The Click Table

A *click table* is a table a Text object consults to determine word boundaries for word selection. It's used whenever the user double-clicks a word, Shift-clicks to extend a word selection, or double-clicks and then drags to select a series of words. The default click table is designed to be used with standard English text. The **setClickTable:** and **clickTable** methods give you control of your Text object's click table.

The Break Table

A *break table* is a table that a Text object uses to determine word boundaries for line breaks. It's similar to the click table, but has a different view of word boundaries. For example, if you insert enough text in the middle of a sentence to push the last word of the sentence onto the next line, the standard break table ensures that both the word and the final punctuation wrap to the next line. In contrast, if you double-click the last word of the sentence, the standard click table ensures that only the word, and not the final punctuation, is selected.

The **setBreakTable:** and **breakTable** methods give you access to the break table, although most programmers will rarely need a break table other than the default one.

Filter Functions

A text object supports two types of input filters: a character filter and a text filter. Character filters remap one character code to another when necessary. Text filters allow for more extensive manipulation of the character or characters to be added to the text.

When the user enters a character from the keyboard, a Text object calls a character filter function to examine the entry. It can also optionally call a text filter function if one is installed. Neither type of filter examines characters that are read into the text from a file or from the pasteboard.

Depending on the current character filter function and the value of the entered character, the Text object adds the character to the list of those to be displayed, interprets the character as a command, or ignores the character.

Unlike a character filter, which only receives the individual character that's entered, a text filter additionally receives information about the state of the Text object. Based on this information, the text filter can change a number of variables including the content of the entered or existing text and the location of insertion point.

Character Filter Functions

The Text class provides two character filters, **NXFieldFilter()** and **NXEeditorFilter()**, and you can write your own if necessary. **NXFieldFilter()** can be used to implement electronic forms. When the user presses Return, Tab, or Shift-Tab, **NXFieldFilter()** causes the Text object to cease being the first responder, letting some other object assume that role. **NXEeditorFilter()** is designed to accept a wider variety of characters. It lets the user enter Tab and Return characters directly into the text.

By default, the Text object uses **NXFieldFilter()** as its character filter. You can change its filter by sending a **setCharFilter** message:

```
[myText setCharFilter:NXEeditorFilter];
```

Character filter functions operate by reassigning the codes of certain characters before sending the code on to the Text object. **NXFieldFilter()** and **NXEeditorFilter()** reassign most codes less than 0x20 (the ASCII space character) to 0x00. Since a Text object ignores input having a value of 0x00, most control characters are excluded from the text.

Codes generated by Return, Tab, and Shift-Tab (back tab) are given special treatment. Depending on the filter, these codes can be sent unaltered to the Text Object or they can be first remapped to the corresponding constant, as defined in **Text.h**:

```
NX_RETURN  
NX_TAB  
NX_BACKTAB
```

When a Text object receives a character code having one of these defined values, it attempts to end its status as first responder. If it has a delegate, the Text object sends the delegate a **textWillEnd:** message, and assuming the delegate allows the change, the Text object then sends a **textDidEnd:endChar:** message. This latter message includes a value that the

delegate can check against these constants to determine which key caused the Text object to lose first-responder status:

```
- textDidEnd:sender endChar:(unsigned short)whyEnd
{
    switch (whyEnd) {
    case NX_RETURN:
        /* send a message based on contents of the Text object */
        break;
    case NX_TAB:
        /* make the next text object the first responder */
        break;
    case NX_BACKTAB:
        /* make the previous text object the first responder */
        break;
    }
}
```

The Matrix class uses code similar to this to let the user tab among the various fields.

NXFieldFilter() is the default character filter function for the Text class. The listing below details how **NXFieldFilter()** remaps the codes generated by Delete, Return, and Tab.

```
NXFieldFilter(unsigned short theChar, int flags)
{
    if (flags & NX_COMMANDMASK)
    {
        theChar = 0;
    } else {
        if (theChar == NX_DELETE)
            theChar = NX_BACKSPACE;
        else if (theChar == NX_CR) {
            theChar = (flags & NX_SHIFTMASK) ? '\n' : NX_RETURN;
        } else if (theChar == '\t') {
            theChar = (flags & NX_SHIFTMASK) ? NX_BACKTAB : NX_TAB;
        } else if (((theChar < ' ') && (theChar != '\n')) &&
                   (theChar != NX_BACKSPACE))
            theChar = 0;
    }
    return (theChar);
}
```

Notice that the codes generated by Return and Tab aren't remapped if Shift is down when the key is pressed. This allows the user to enter these characters even if **NXFieldFilter()** is in use.

The Text class also provides **NXEeditorFilter()**, a character filter function for more general text editing:

```

NXEditorFilter(unsigned short theChar, int flags)
{
    if (flags & NX_COMMANDMASK)
    {
        theChar = 0;
    } else {
        if (theChar == NX_DELETE)
            theChar = NX_BACKSPACE;
        else if (theChar == NX_CR) {
            theChar = '\n';
        } else if (theChar == '\t') {
            ;
        } else if ((theChar < ' ') && (theChar != '\n') &&
                   (theChar != NX_BACKSPACE))
            theChar = 0;
    }
    return (theChar);
}

```

Notice that **NXEditorFilter()** passes the codes generated by Return and Tab through to the Text object. Shift has no effect on how these codes are handled.

You might want to write your own character filter function. For example, **Text.h** defines four other constants that cause a Text object to attempt to resign first responder status:

```

NX_LEFT
NX_RIGHT
NX_UP
NX_DOWN

```

By writing a character filter function that remaps certain Control sequences to these constants (such as Control-H, or 0x08, to NX_UP), you could provide alternate movement commands. Alternatively, with the help of the Text object's delegate, the new character filter could call a help system whenever the user entered such a Control sequence.

Text Filter Functions

The Text class provides for a text filter function although, by default, a text object doesn't use one. To install a text filter function, send a **setTextFilter:** message:

```
[myText setTextFilter:FilterText];
```

The **textFilter** method returns the current text filter.

Once a text filter function is installed, it's called whenever text is entered from the keyboard. The filter receives four arguments: the **id** of the caller, a pointer to the character being entered, a pointer to the length of the text to be inserted, and the character position of the insertion. The filter can alter the inserted text, or the Text object's preexisting text, in any way. Whatever the alteration, the text filter must inform the Text object of new text to be inserted and the length of the inserted matter.

For example, the following simple text filter examines incoming characters for the occurrence of the “&” character. If it finds one, it then checks the character immediately preceding the current character position. If the preceding character is a space, it replaces the “&” with the word “and”. Note that if it makes such a substitution, it returns the length of the substitute string in the address referred to by **inputLength**:

```
char sample[3];
char model[] = " &";
char substitute[] = "and";

char *FilterText(id textObj, char *inputText, int *inputLength,
                 int position)
{
    if (position > 0) {
        /* get character that precedes this one */
        [textObj getSubstring:sample start:position-1 length:1];

        /* add this character and null terminator */
        sample[1] = *inputText;
        sample[2] = '\0';

        /* compare sample with model */
        if (!strcmp(sample, model)) {
            /* make substitution if they are equal */
            *inputLength = 3;
            return (substitute);
        }
    }
    /* otherwise, simply return original character */
    return (inputText);
}
```

A Text Object in a Scrolling View

To accommodate large blocks of text, a Text object is commonly made a subview of a ScrollView. In this way, the user can scroll into view portions of the text that lie beyond the ScrollView’s frame. A Text object must, however, be properly configured before it will cooperate with the ScrollView.

This example program creates a ScrollView within a Window and installs a Text object as the ScrollView’s document view:

```
#import <appkit/appkit.h>

main(int argc, char *argv)
{
    id theWindow, theScrollView, myText;
    NXRect aRect, contentRect;
    NXSize aSize;
```

```

NXApp = [Application new]; /* create Application object */

NXSetRect (&aRect, 100.0, 350.0, 300.0, 300.0);
theWindow = [Window newContent:&aRect /* create Window */
             style:NX_TITLEDSTYLE /* object */
             backing:NX_BUFFERED
             buttonMask:NX_ALLBUTTONS
             defer:NO];
[theWindow setBackgroundGray:NX_WHITE];

theScrollView = [ScrollView newFrame:&aRect]; /* create */
[theScrollView setVertScrollerRequired:YES]; /* a */
[theScrollView setHorizScrollerRequired:NO]; /* ScrollView*/

[theWindow setContentView:theScrollView];

contentRect = aRect; /* find size of content view */
[ScrollView getContentSize:&(contentRect.size)
    forFrameSize:&(aRect.size)
    horizScroller:NO
    vertScroller:YES
    borderType:NX_NOBORDER];

myText = [Text newFrame:&contentRect /* create a Text */
          text:NULL /* object */
          alignment:NX_LEFTALIGNED];
[myText notifyAncestorWhenFrameChanged:YES]; /*configure*/
[myText setVertResizable:YES]; /* it to work with */
[myText setHorizResizable:NO]; /* the ScrollView */

aSize.width = 0.0; /* let the Text object get no smaller */
aSize.height = contentRect.size.height; /* than height of */
[myText setMinSize:&aSize]; /* ScrollView's content view */

aSize.width = contentRect.size.width; /* let the Text */
aSize.height = 1000000; /* object's height grow */
[myText setMaxSize:&aSize]; /* to large value */

[theScrollView setDocView:myText]; /* make myText the doc.*/
[[myText superview] setAutoresizingMask:YES]; /* view */
[[myText superview] setAutosizing:NX_HEIGHTSIZABLE |
    NX_WIDTHSIZABLE]; /* notify Text object if Window resized */

[theWindow display]; /* display the window in */
[theWindow orderFront:nil]; /* front of other windows; */
[theWindow makeKeyWindow]; /* make it the key window */

[NXApp run]; /* start the Application */
}

```

In this example, a Text object is made the document view of a ScrollView. The ScrollView only allows scrolling in the vertical direction, so the Text object is also constrained to grow only vertically. The Text object's frame is sized to fit within the scrolling portion of the

ScrollView. When a user adds enough text to the Text object to cause its frame to grow, the **notifyAncestorWhenFrameChanged:** message ensures that the ScrollView is notified of the change. The ScrollView automatically scrolls the new line of text into view and then displays a knob to allow the user to access other portions of the document view. The knob will disappear if the user deletes enough text to shrink the Text object to its original, and minimum, size.

Reusing a Text Object

As an efficiency, a single Text object can be reused: It can be made to draw in various Views or various locations within one View. In fact, you can make use of a Window's field editor, the Text object that the Application Kit uses to draw text within standard Kit objects. Using one Text object rather than several saves both startup time and memory.

Archiving a Text Object

The **write:** method writes the Text object's instance variables out to the archive stream. It doesn't, however, maintain the values of the following instance variables if you've reassigned their values to functions or tables not provided by the Text class:

```
charFilterFunc  
scanFunc  
drawFunc  
charCategoryTable  
preSelSmartTable  
postSelSmartTable  
breakTable  
clickTable
```

The **read:** method reads the Text object's instance variables into memory from the archive stream. Once the Text object is reestablished in memory, you may want to reset some of the values of the instance variables above.

One way to do this is to send the appropriate messages to the Text object from another object's awake method. **awake** messages are only sent after the Text object and all the objects it refers to have been read in, ensuring that the Text object is ready to receive the initializing messages. In the same way, you can send a **setSel::** message to reestablish the selection.

Rich Text Format Support

A Text object can encode and decode a subset of the formatting commands defined in RTF; the following table summarizes that support. For example, a Text object can read and properly display the characters associated with a **\b** control word, making those characters have a bold attribute. It can also embed this control word along with the affected characters when it writes the file to the disk. For other control words, such as **\margln**, a Text object will display the associated characters properly but won't write the control word when it writes the characters to a file. When reading text from a file, a Text object ignores any RTF control word that's not listed in the following table.

Control Word	Read/Write
\ansi	yes/yes
\paperwn	yes/yes
\margln	yes/yes
\margrn	yes/yes
\pard	yes/no
\sn	yes/no
\ql	yes/yes
\qr	yes/yes
\qc	yes/yes
\fin	yes/yes
\lin	yes/yes
\b	yes/yes
\i	yes/yes
\fn	yes/yes
\fsn	yes/yes
\upn	yes/yes
\dn	yes/yes
\par	yes/yes
\tab	yes/yes

The Text class doesn't write the **\pard** control word; instead, it manipulates groupings to restore paragraph default values when needed. In addition, the Text class interprets the **\sect**, **\page**, and **\line** control words as Return characters. The Text class recognizes all installed fonts. See *Rich Text Format Specification* by Microsoft Corporation for more information about RTF.

The Box Class

A Box is a View that visually groups other Views. As shown in Figure 9-3, a typical Box displays a border and a title. Box objects are often used to group choices that a user can make in a panel. For example, in a Find panel, one Box may surround buttons that set the scope of the search; another may enclose the switches that specify whether the case of letters should be ignored and whether the text to be found should be treated as a regular expression.

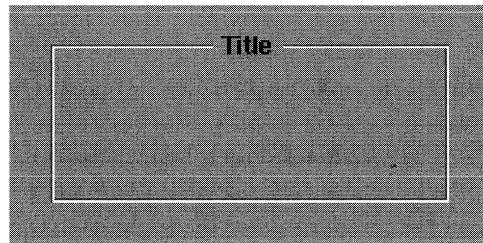


Figure 9-3. A Typical Box

A Box's border encloses its contents. After changing the Box's contents, you can resize the Box to fit. You can also change how the Box looks by modifying its border type and the title's position and font. The following two sections discuss how to accomplish these tasks. However, you may want to use Interface Builder to create and set the initial characteristics of a Box, since that's easy to do. Your application code could then use the methods discussed in the following sections to modify the Box dynamically at appropriate moments in your program.

Creating and Modifying a Box Object

The **newFrame:** class method returns a new instance of the Box class with the location, height, and width specified by its argument:

```
NXRect myRect;
NXSetRect (&myRect, 200.0, 200.0, 500.0, 500.0);
myBox = [Box newFrame:&myRect];
```

Since you'll probably be adding to the Box's initially empty contents and then resizing the Box, its original size isn't critical. It's easy to make the Box larger or smaller as needed to encompass its contents. The "Resizing the Box" section below explains how to do this.

A Box object defines a set of rectangles that determine how component parts are laid out. These rectangles are shown in Figure 9-4.

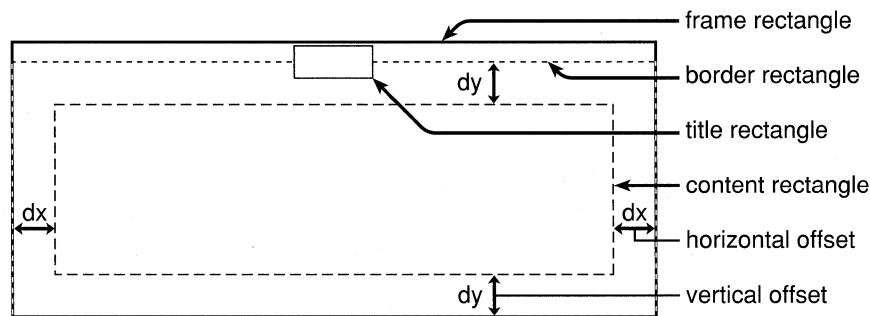


Figure 9-4. The Layout Rectangles of a Box

The external boundary of the Box is defined by its frame rectangle. Within this boundary lies the border rectangle; its location is determined by the location and size of the title rectangle, the rectangle that encloses the title. If you choose to display the title above the top edge of the border, for example, you will cause the border rectangle to shrink. The innermost, or content, rectangle encompasses the contents of the Box. You can set the value of the offsets, which determine the amount of horizontal and vertical space between the content rectangle and the border rectangle.

The Border

A Box's border can be marked with a line, a bezel, a groove, or nothing at all. Figure 9-5 shows these border types and the corresponding constants you can use to set the type.

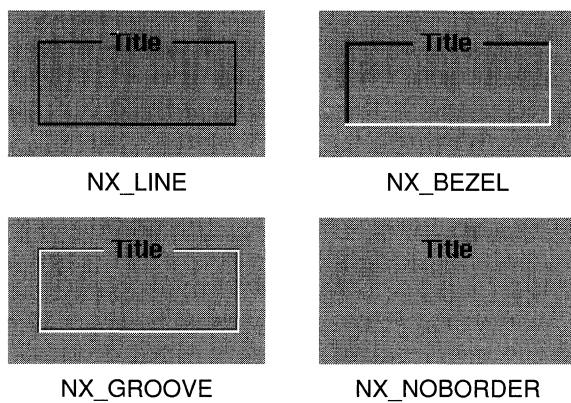


Figure 9-5. Border Types

By default, a Box has a grooved border. To change this, you send a **setBorderType:** message to the Box, with one of the four constants as its argument. For example, to set the border type to a bezel:

```
[myBox setBorderType:NX_BEZEL];
```

You can also query the Box about its border type, with the **borderType** method:

```
int myBorderType;  
myBorderType = [myBox borderType];
```

Since the border is drawn inside the Box, it will slightly reduce the space available for the content rectangle. A borderless Box doesn't affect the content rectangle, but lines, bezels, and grooves reduce both the width and the height of the content rectangle by 2.0, 4.0, and 6.0, respectively. In addition to setting the border type, the **setBorderType:** method recalculates the size of the content rectangle based on the width of the new border.

The Title

You can have a Box with no title or one with a title positioned above, below, or intersected by the border at either the top or the bottom of the Box. Figure 9-6 shows the possible title positions and the corresponding constants you can use to set the position. The default position is NX_ATTOP.

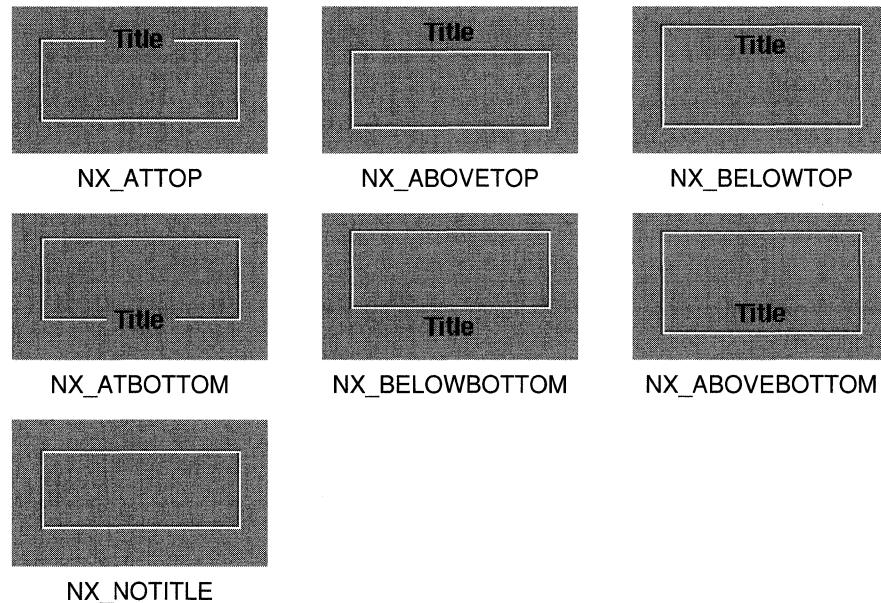


Figure 9-6. Title Positions

You set the title position by specifying one of the seven constants with the **setTitlePosition:** method. The following example sets the position to be above the top of the border:

```
[myBox setTitlePosition:NX_ABOVETOP];
```

The **setTitlePosition:** method adjusts the rectangles of the Box so that they're positioned correctly relative to the title.

A Box's default title is "Title". To change this, you send the Box a **setTitle:** message and specify the text you want displayed as the title:

```
const char *myTitle ="Find Options";
[myBox setTitle:myTitle];
```

In addition to changing the text, the **setTitle:** method recalculates the size of the title rectangle. If the title is longer than the width of the Box, the title rectangle's width is reduced to the size of the Box and the text is clipped. **setTitle:** also provides a small cushion of space on the top and bottom of the title so that it doesn't actually touch the Box's border.

By default, the title will be displayed using 12-point Helvetica. However, you can use the **setFont:** method to specify a new font, as shown below. (You could also use the defaults mechanism to change the system font, but this may affect more than the title of your Box; see “The Defaults System” in Chapter 10, “Support Objects and Functions.”)

```
[myBox setFont:aNewFontObj];
```

The **setFont:** method recalculates the Box’s rectangles. If you increased the font size significantly, for example, the title rectangle would increase; the content rectangle and possibly the border rectangle (if the title is outside of or intersected by the border) would decrease to accommodate the larger title rectangle.

You can also query a Box about its title, title position, and font:

```
id      myFontObject;
char   *myTitle;
int    myTitlePosition;

myTitle = [myBox title];
myTitlePosition = [myBox titlePosition];
myFontObject = [myBox font];
```

The **title** method returns a pointer to the Box’s title. **titlePosition** returns one of the seven constants that correspond to the title’s position, and **font** returns the **id** of the font object used to display the title.

The Offsets

The content rectangle of a Box can be horizontally and vertically offset inside the border. This allows you to adjust the space between the contents and the border and to make the horizontal offset different than the vertical offset. As shown above in Figure 9-4, the two vertical offsets (at the top and bottom of the Box) are equal, and the horizontal offsets at each side are equal. When you set the offsets, the size of the Box rather than the content rectangle is changed to accommodate the offsets.

Initially, a Box is created with default offsets of 5.0. To set new offsets, you supply them as arguments to the **setOffsets:** method, as shown below. (Although you can specify negative offset values, the contents of the Box or the Box itself may be clipped.)

```
[myBox setOffsets:40.0 :0.0];
```

The first argument for the **setOffsets** method refers to the horizontal offset, and the second refers to the vertical offset. In this example, if the previous offset values were the default 5.0, the horizontal dimension of the Box’s frame rectangle will increase by a total of 70.0 and the vertical dimension will decrease by 10.0. After changing the offsets, you’ll want to resize the box using the **setFrameFromContentFrame** method, which is described in more detail in the “Resizing the Box” section below.

You can obtain the current values of a Box's offsets by querying the Box:

```
NXSize mySize;  
[myBox getOffsets:&mySize];
```

The offsets will be placed in **mySize**, which must be an NXSize structure.

The Contents of the Box

As a graphic object, a Box's purpose is to visually group its contents, typically by displaying a border and a categorizing title. As discussed above, you can modify the border and title in several ways. You can also add to the Box's contents and then resize it so that it comfortably encloses them. For example, you may want to use a Matrix of switches that changes the number of Cells it displays. The next two sections discuss how to manage a Box's contents and how to resize the Box.

Box's View Hierarchy

Views are added to the Box's contents by making them subviews. Here **newSubview** is added to **myBox**:

```
[myBox addSubview:newSubview];
```

This actually makes **newSubview** a subview of the content view, which corresponds to the Box's content rectangle. The content view groups the Box's contents and makes sure that they're displayed only within the content rectangle. Since the content view is a subview of Box, the frame rectangles of Views added to the Box should reflect their position in the content rectangle, not the Box's frame rectangle. After you've added a subview, you'll probably want to use the **sizeToFit** method (described below) to adjust the Box's size to accommodate its new subview.

To replace the existing content view, you send a **setContentView:** message to the Box, giving it the new content view as the argument:

```
[myBox setContentView:newContentView];
```

The old content view is returned so that you can either free it or use it in another view hierarchy. Since **setContentView:** recalculates the size of the Box based on the size of the new content view, you don't need to resize it.

Resizing the Box

If you've added a subview to the Box's contents, you should use the **sizeToFit** method to adjust the size of the Box. **sizeToFit** calculates the appropriate size for the content rectangle so that it just encloses all the content view's subviews. It then uses the **setFrameFromContentFrame:** method to resize the Box to match the new content rectangle.

You can also send the **setFrameFromContentFrame:** method directly to the Box to specify a new location and size for the content rectangle:

```
NXRect myContentRectangle;  
  
NXSetRect (&myContentRectangle, 100.0, 100.0, 300.0, 300.0);  
[myBox setFrameFromContentFrame:&myContentRectangle];
```

The Box is resized and relocated so that its content view has the same dimensions as the specified content rectangle, which is in the coordinate system of the Box's superview.

You can also resize the Box from the outside in, using the **sizeTo::** method. The two arguments specify the new width and height of the Box. The Box's layout rectangles are then recalculated to fit inside this new boundary. If the new size of the Box is too small for its content view (taking into account the offsets), the drawing that the content view displays will be clipped at the Box's boundaries.

Chapter 10

Support Objects and Functions

10-4 Streams

- 10-4 Writing and Reading
- 10-5 Writing and Reading Characters
- 10-5 Writing and Reading Bytes of Data
- 10-6 Writing and Reading Formatted Data
- 10-6 Flushing and Filling
- 10-7 Seeking
- 10-7 Connecting Streams to a Source or Destination
- 10-8 Connecting to Memory
- 10-8 Connecting to a File
- 10-10 Connecting to a Mach Port

10-11 Archiving to a Typed Stream

- 10-11 Archiving a Data Structure
- 10-12 Archiving Arbitrary Data
- 10-13 Archiving Arrays and NXPoint, NXSize, and NXRect Structures
- 10-14 Archiving Objects
 - 10-14 The **write:** Method
 - 10-15 Archiving **id** Instance Variables
 - 10-16 Archiving an Object with **id** Instance Variables
- 10-16 Unarchiving a Data Structure
- 10-17 Unarchiving Objects
 - 10-17 The **read:** Method
 - 10-18 Initializing an Object
- 10-19 Opening and Closing a Typed Stream
- 10-19 A Typed Stream on a File
- 10-20 A Typed Stream on Memory
- 10-20 Using an NXStream Structure

10-21 The Defaults System

- 10-21 Creating a Registration Table
- 10-23 Reading Default Values
- 10-23 Writing Default Values
- 10-25 Changing the Defaults Database from a Shell Window
- 10-25 The Command Line
- 10-26 System and Global Parameters
 - 10-26 System Parameters
 - 10-27 Global Parameters
- 10-30 Workspace Manager's Parameters

10-31 The Pasteboard

- 10-31 Using the Pasteboard
- 10-31 Declaring Data Types
- 10-32 Copying Data to and Reading it from the Pasteboard
- 10-33 Examples of Preparing and Parsing Data
- 10-33 Using a Stream
- 10-34 Using a Typed Stream
- 10-36 Responding to Cut, Copy, and Paste

10-36 Exception Handling

- 10-37 Detecting Exceptional Conditions
- 10-39 Raising an Exception
- 10-39 Handling an Exception
- 10-42 Nested Exception Handlers
- 10-43 Raising an Exception Outside of an Exception Handler
- 10-43 Exception Codes
 - 10-44 Defining Codes for Your Application
 - 10-45 Associating Messages with Codes
 - 10-45 Reporting Errors
 - 10-46 Registering Error Reporters
 - 10-47 Handling PostScript Errors
 - 10-49 Managing Exception Data

Chapter 10

Support Objects and Functions

Changes made for the current release of NeXTstep affect the information presented in this chapter. For details see:

[/NextLibrary/Documentation/NextDev/ReleaseNotes/AppKit.rtf](#)

In addition to a program structure for applications that use the NeXT window system and a variety of preprogrammed user-interface objects, the Application Kit offers a number of other program support facilities. Some are implemented as class definitions and some as standard C functions and macros. All are designed to work well with the Kit's program structure and user-interface objects. They include:

- A set of functions for writing and reading data to streams.
- A set of functions that allow you to save data structures, including objects, in an archive file and load them from the file into an application.
- A system for specifying program defaults.
- A Pasteboard object that supports cut, copy, and paste operations.
- A Font and a FontManager object that help applications get information about a specific font and serve as a vehicle for setting the font in the Window Server.
- List, Storage, and HashTable objects that act as general memory allocators. The StreamTable object is a specialized storage class.
- A mechanism for handling errors.

Other Application Kit objects depend on these support facilities. For example, the Text class uses a variety of Font objects and the Pasteboard for cut and paste operations. Program support facilities aren't confined to a behind-the-scenes role, however. You can make direct use of them in your program.

Streams

A stream is a sequence of data into or out of a program. It acts as a channel, connecting an application with a source of data or a destination for data. If you use a stream (rather than a file, for example) for the input or output of data, you can read or write data without regard to its source or destination. For example, suppose you designed an object that writes data to a stream. You can save that data in a file, or send it to another process using a Mach port, without changing the object. You only need to call the function that connects a stream to a file or a Mach port and then pass the connected stream to the object.

The Application Kit writes instances of its classes to a special kind of stream, a typed stream. Typed streams are particularly useful for writing and reading objects and other complex data structures. See “Archiving to a Typed Stream” later in this chapter for more information.

When using a stream, you can select memory, a file, or a Mach port as the source or destination by calling the appropriate function to create the stream. You can also implement the functions needed to connect a stream to a different source or destination, such as a Text object, thereby creating your own type of stream.

Memory and file streams allow two-way data flow—that is, you can use the same stream for both writing and reading. In addition, you can set the position of the next input or output operation on these streams. For example, you can read the first few bytes of data from a memory stream, skip to the middle to read some more, and then write data at the end of the stream.

The next section discusses how to write data to and read it from a stream. Then the steps needed to connect the stream to memory, a file, or a Mach port are presented.

Writing and Reading

The functions that write to or read from a stream can be grouped into three categories, depending on whether they:

- Write or read single characters at a time,
- Write or read a specified number of bytes of data, or
- Convert data according to a format string as it's read or written.

These functions are modeled after the standard C library functions for input and output. If you are familiar with those standard C functions, you know in a general way what the corresponding NeXT-defined functions do.

The NeXT functions for writing and reading take a pointer to a stream as an argument. These functions can be used with a stream connected to any source or destination. In the examples shown below, this stream has already been connected and is referred to as **stream**. See the section “Connecting Streams to a Source or Destination” below for details on connecting a stream.

Writing and Reading Characters

The macros for writing and reading single characters at a time are similar to the corresponding standard C functions: **NXPutc()** and **NXGetc()** work like **putc()** and **getc()**. **NXPutc()** appends a character to the stream:

```
NXPutc(stream, 'c');
```

The second argument specifies the character to be written to the stream. **NXGetc()** retrieves the next character from the stream:

```
unsigned char aCharacter;
aCharacter = NXGetc(stream);
```

The **unsigned char** type should be used for portability.

To reread a character, call **NXUngetc()**. This function puts the last character read back onto the stream:

```
unsigned char aCharacter;

NXUngetc(stream);
aCharacter = NXGetc(stream);
```

Note that **NXUngetc()** doesn't take a character as an argument as **ungetc()** does. **NXUngetc()** can only be called once between any two calls to **NXGetc()** (or any other reading function).

Writing and Reading Bytes of Data

The functions **NXWrite()** and **NXRead()** write multiple bytes of data to and read them from a stream. In the following example, an **NXRect** structure is written to a stream.

```
NXRect myRect;

NXSetRect(&myRect, 0.0, 0.0, 100.0, 200.0);
NXWrite(stream, &myRect, sizeof(NXRect));
```

The second and third arguments for **NXWrite()** give the location and amount of data (measured in bytes) to be written to the stream. To read data from a stream, call **NXRead()**:

```
NXRect myRect;
NXRead(stream, &myRect, sizeof(NXRect));
```

NXRead() reads the number of bytes specified by its third argument from the given stream and places the data in the location specified by the second argument.

Writing and Reading Formatted Data

Four functions convert strings of data as they're written to or read from a stream. **NXPrintf()** and **NXScanf()** take a character string that specifies the format of the data to be written or read as an argument. **NXPrintf()** interprets its arguments according to the format string and writes them to the stream. Similarly, **NXScanf()** reads characters from the stream, interprets them as specified in the format string, and stores them in the locations indicated by the last set of arguments. The conversion characters in the format string for both functions are the same as those used for the standard C library functions, **printf()** and **scanf()**. The examples below illustrate the use of some of these conversion characters. For detailed information on these characters and how conversions are performed, see the UNIX manual pages for **printf()** and **scanf()**.

The following writes data of the form “Please send 500 bucks before Friday” to a stream:

```
int      amt = 500;
char    *day = "Friday";

NXPrintf(stream, "Please send %d bucks before %s", amt, day);
```

The call

```
int      numint;
float   numflo;
char    name[15];

NXScanf(stream, "%d%f%s", &numint, &numflo, name);
```

with the stream of data

```
5 19.61 Jacqueline
```

will assign 5 to the variable **numint**, 19.61 to **numflo**, and “Jacqueline” to **name**.

Two related functions, **NXVPrintf()** and **NXVScanf()**, are exactly the same as **NXPrintf()** and **NXScanf()**, respectively, except that instead of being called with a variable number of arguments, they are called with a **va_list** argument list, which is defined in the header file **stdarg.h**. This header file also defines a set of macros for advancing through a **va_list**.

Flushing and Filling

File and Mach port streams are buffered, which means that data is initially written to a buffer rather than to the file or the port itself. If you write more data than the buffer can hold, the buffer is flushed, sending all the data to the destination that the stream is connected to. Also, before a stream is disconnected from its destination, the buffer is flushed to ensure

that all data actually gets sent to the destination. Usually you won't need to flush the buffer yourself. However, if you don't want to disconnect the stream but your code depends on knowing that all data has been sent to the stream's destination, call **NXFlush()**:

```
NXFlush(stream);
```

If **NXFlush()** is called with a memory stream, more memory is made available for writing. However, you don't need to call this function with a memory stream since more memory is automatically allocated as needed.

When reading from a file or Mach port stream, data is loaded into a buffer and then read from the buffer. This buffer is automatically filled after you've read all the data in it. To explicitly fill the buffer yourself, call **NXFill()**:

```
NXFill(stream);
```

Calling this function with a memory stream has no effect.

Seeking

Stream functions for writing and reading start at the current position of the stream, so you may need to manipulate the position of the stream:

```
NXSeek(stream, 0, NX_FROMSTART);
```

NXSeek() moves forward by the number of bytes specified by its second argument relative to the position indicated by its last argument, which can be **NX_FROMSTART**, **NX_FROMCURRENT**, or **NX_FROMEND**. In the example, the current position is set to the beginning of the stream. The function **NXTell()** returns an **int** that specifies the current position in the stream given as its argument. This value, which is measured in bytes from the beginning of the stream, can be used in a call to **NXSeek()**.

Position within some streams—for example, Mach port streams—is undefined, so these two functions shouldn't be used with such streams. They can be used with memory or file streams, but they should be avoided if the stream will ever be connected to an unseekable source or destination. The **NX_CANSEEK** flag, defined in the header file **streams/streams.h**, indicates whether a stream is seekable.

Connecting Streams to a Source or Destination

Functions are provided to open a stream on memory, a file, or a Mach port. Opening a stream involves connecting it to a source or destination and specifying whether it will be used for writing or reading (or both). Regardless of what the source and destination are, you can use the functions described above for writing data to and reading it from the stream.

When you're finished with the stream, use the appropriate function to close it. The functions for closing a stream disconnect it from its source or destination and release storage used by the stream. (The NXStream structure used below is defined in the header file **streams/streams.h**.)

Connecting to Memory

A memory stream is a temporary buffer for writing or reading data. To open a memory stream, call **NXOpenMemory()**:

```
NXStream *stream;  
stream = NXOpenMemory(NULL, 0, NX_WRITEONLY);
```

If **NX_WRITEONLY** is specified, the first two arguments should be **NULL** and **0** to allow the amount of memory available to be automatically adjusted as more data is written. If **NX_READONLY** is specified, a memory stream will be set up for reading the data beginning at the location specified by the first argument. The second argument indicates how much data will be read. To use the stream for both writing and reading, you can either use **NULL** and **0** or specify the location and amount of data to be read.

When you're finished with a memory stream, close it by calling **NXCloseMemory()**:

```
NXCloseMemory(stream, NX_FREEBUFFER);
```

Usually, you'll use **NX_FREEBUFFER** as the second argument to free all storage used by the stream, but there are two other constants that can be used. If you've used the stream for writing, more memory may have been made available than was actually used; the constant **NX_TRUNCATEBUFFER** indicates that any unused pages of memory should be freed. (Calling **NXClose()** with a memory stream is equivalent to calling **NXCloseMemory()** and specifying **NX_TRUNCATEBUFFER**.) **NX_SAVEBUFFER** doesn't free the memory that had been made available.

Before you close a memory stream, you can save data written to the stream in a file. To do this, call **NXSaveToFile()**, giving it the stream and a pathname as arguments:

```
const char *home = NXHomeDirectory();  
NXSaveToFile(stream, home);
```

NXSaveToFile() writes the contents of the memory stream into the file, creating it if necessary. After saving the data, close the stream using **NXCloseMemory()**.

Connecting to a File

Two functions are available to connect a stream to a file. **NXMapFile()** maps a file into memory and then opens a memory stream; **NXOpenFile()** connects a stream to the file. Memory mapping allows efficient random and multiple access of the data in the file, so

NXMapFile() should be used whenever the file is stored on disk. (The *NeXT Operating System Software* manual discusses memory mapping in more detail.) If you want to connect a stream to a pipe or a socket, use **NXOpenFile()**. This function takes a file descriptor as an argument.

To map a file into memory, call **NXMapFile()**, giving it the pathname for the file and indicating whether you'll be writing, reading, or both:

```
NXStream *stream;
stream = NXMapFile("aPathname", NX_READONLY);
```

This function opens a memory stream and initializes it with the contents of the file. Then you can use the functions described above for writing and reading. If you use the stream only for reading, just close the memory stream when you're finished. If you write to the stream, you need to explicitly save the data written before closing:

```
NXSaveToFile(stream, "aPathname");
NXCloseMemory(stream, NX_FREEBUFFER);
```

To open a file stream using a file descriptor, call **NXOpenFile()**, giving it the descriptor and specifying how the stream will be used:

```
NXStream *stream;
stream = NXOpenFile(fd, NX_WRITEONLY);
```

If the file descriptor was obtained through the system call **open()**, one of three flags (O_WRONLY, O_RDONLY, or O_RDWR) was used to open the file for writing, reading, or both. The meaning of this flag must match that used in the call to **NXOpenFile()**. See the UNIX manual page on **open()** for more information about its arguments.

You can use **NXOpenFile()** to connect to **stdin**, **stdout**, and **stderr** by obtaining their file descriptors using the standard C library function **fileno()**. (For more information on this function, see its UNIX manual page.) The following example allows you to read from **stdin**.

```
int      fileDescriptor;
NXStream *stream;

fileDescriptor = fileno(stdin);
stream = NXOpenFile(fileDescriptor, NX_READONLY);
```

After you've finished with the file stream, you need to disconnect it from the file and free the storage used by the stream:

```
NXClose(stream);
```

NXClose() saves any data you wrote to the stream in the file, but it doesn't release the file descriptor. To release the descriptor, use the system call **close()**, giving it the descriptor as an argument.

Connecting to a Mach Port

In Mach, tasks and threads communicate among themselves and with the operating system kernel by sending messages. These messages must adhere to a certain structure. They're sent to a Mach port using the Mach function **msg_send()**, where they're queued until read by the receiver using **msg_receive()**. Rather than setting up this message structure yourself, you can connect a stream to a Mach port using **NXOpenPort()**. Then when you use **NXWrite()** or **NXRead()** (depending on whether you are sending or receiving data), the data you send or receive will be sent to or dequeued from the port you specify. Mach ports and messages are described in more detail in the *NeXT Operating System Software* manual.

Mach port streams can't be opened for both writing and reading, so you need to connect one stream to a port to send data and another stream to the same port to read that data. The following paragraphs show how to set up such a pair of streams.

To send data to a Mach port, first open a stream using **NXOpenPort()**. A port must be previously allocated using the Mach function **port_allocate()**; see the *NeXT Operating System Software* manual for more information about using this function.

```
port_t      thePort;
NXStream   *outStream;

outStream = NXOpenPort(thePort, NX_WRITEONLY);
```

Since this is the sender's stream, it's opened for writing. Now, using any of the functions for writing described above, write to the opened stream the data you want to send to the port. You should close the stream after you finish writing to it:

```
NXClose(outStream);
```

This ensures that all data is actually sent to the port by flushing the buffer associated with the stream. If you want to keep the stream open, you can flush the buffer using **NXFlush()**:

```
NXFlush(outStream);
```

To read this data, the receiver opens a stream on the same port for reading. This can be done independently of when the sender's stream was opened and when the data was sent.

```
port_t      thePort;
NXStream   *inStream;

inStream = NXOpenPort(thePort, NX_READONLY);
```

The functions for reading data will wait until it's available and then read it into the specified location. After the data has been read, the stream can be closed:

```
NXClose(inStream);
```

Archiving to a Typed Stream

Archiving is the process of preserving a data structure, especially an object, for later use. An archived data structure is usually stored in a file, but it can also be written to memory, copied to the pasteboard, or sent to another application. Archiving involves writing data to a special kind of data stream, called a *typed stream*. During unarchiving, memory is allocated for the data structure, and it's initialized with values read from a typed stream.

Typed streams are an abstraction built on the streams abstraction discussed earlier in this chapter. Because of this relationship, data can be written to and read from a typed stream without regard to what destination or source the stream is connected to. Once you've written the code to archive a data structure, you can use that code to store the data structure in a file, write it to memory, or send it to a Mach port.

Typed streams are used for archiving because they provide some protection against future changes that might affect the ability to unarchive a data structure. When a typed stream is used, the data type is archived along with the data and, in the case of objects, the object's class hierarchy and version are also archived. This additional information is checked when a data structure is unarchived, and an exception is raised if necessary. Typed streams also provide some degree of data portability between machines.

The archiving functions make it easy to write structures consisting of several different data types, including objects. Archiving with a typed stream also ensures that objects are written only once even if several members of a data structure refer to the same object. In addition, when archiving an object, you can limit the scope of what's archived by deciding which objects referred to by `id` instance variables should be archived. Classes defined in the Application Kit and the common classes archive themselves using typed streams. If you include instances of these classes (or subclasses) in a data structure, you'll want to archive it using typed streams.

All functions mentioned in this section are described in the *NeXTstep Reference* manuals.

Archiving a Data Structure

To write data to a typed stream, you use any of several functions. Two of these, `NXWriteType()` and `NXWriteTypes()`, allow you to specify the data type or types being written. Other functions write a specific data type; for example, `NXWritePoint()` writes an `NXPoint` structure. The archiving functions are listed below and discussed in the following sections. The functions for unarchiving are similar to these, and they're listed and described later.

Function	Data Type
NXWriteType()	Single specified type
NXWriteTypes()	Multiple specified types
NXWriteArray()	An array
NXWritePoint()	An NXPoint structure
NXWriteSize()	An NXSize structure
NXWriteRect()	An NXRect structure
NXWriteObject()	An id
NXWriteObjectReference()	An id

All these functions take a pointer to a typed stream as their first argument. Since you can write to a typed stream without knowing what it's connected to, opening a typed stream and writing to it are described separately. The "Opening and Closing a Typed Stream" section below explains how to obtain a typed stream pointer.

Archiving Arbitrary Data

NXWriteType() and **NXWriteTypes()** write strings of data to a typed stream. (These functions are similar to the **printf()** standard C function, which is described in its UNIX manual page.) They take a pointer to a typed stream, a character string indicating the format of the data to be read or written, and the address of the data as arguments. The format string characters and their corresponding data types are listed below. They're described in more detail in *NeXTstep Reference, Volume 2*.

Format Character	Data Type
c	char
s	short
i	int
f	float
d	double
@	id
*	char *
%	NXAtom
:	SEL
#	class
!	int; corresponding data won't be read or written
{ <i>type</i> }	struct
[<i>count type</i>]	array

NXWriteType() writes data as the single data type specified by its format string. **NXWriteTypes()** writes multiple types of data. The types are listed in the format string using the appropriate format characters shown above, and pointers to matching data are listed as the last arguments. This example shows three different data types being written to a typed stream:

```

float    aFloat = 3.0;
int     anInt = 5;
char   *aCharStar = "foo";

NXWriteTypes(typedStream, "fi*", &aFloat, &anInt, &aCharStar);

```

If **NXWriteType()** had been used, three lines of code would have been necessary, one for each data type. Both functions take pointers to the data to be written, unlike **printf()**; this implementation results in the corresponding archiving and unarchiving functions taking the same arguments.

Both functions are particularly useful for writing structures consisting of several kinds of data. For example, this structure

```

typedef struct {
    float aFloat;
    int anInt;
    char *aCharStar;
} MyStruct;

```

would be written as follows:

```
NXWriteType(typedStream, "{fi*}", &MyStruct);
```

Both **NXWriteType()** and **NXWriteTypes()** write objects if the "@" format character is used, which is equivalent to calling **NXWriteObject()**. The section "Archiving Objects" below explains the issues involved in writing objects and the different ways of archiving them.

Archiving Arrays and NXPoint, NXSize, and NXRect Structures

For convenience, several functions are provided to archive specific kinds of data structures. These structures can all be written using **NXWriteType()** or **NXWriteTypes()**, but it's easier to use the specialized functions.

NXWriteArray() writes an array to the typed stream passed as its first argument. You specify the number of elements in the array and their type. The following is an example of an integer array being written.

```

int myArray[4];

myArray [0] = 0; myArray [1] = 11;
myArray [2] = 22; myArray [3] = 33;
NXWriteArray(typedStream, "i", 4, myArray);

```

Note that **NXWriteArray()** takes an array, not a pointer to an array, as an argument.

NXWritePoint(), **NXWriteSize()**, and **NXWriteRect()** work through **NXReadType()** to write NXPoint, NXSize, or NXRect structures to a typed stream. The following example shows these three data structures being archived.

```
NXPoint zeroPoint = {0.0, 0.0};  
NXSize rectSize = {100.0, 200.0};  
NXRect aRect = {zeroPoint, rectSize};  
  
NXWritePoint(typedStream, &zeroPoint);  
NXWriteSize(typedStream, &rectSize);  
NXWriteRect(typedStream, &aRect);
```

Archiving Objects

Archiving an object begins with a call to either **NXWriteRootObject()** or **NXWriteObject()**. These functions take a pointer to a typed stream and an object’s **id** as arguments. They send the object a **write:** message, passing it the typed stream. The **write:** method contains the code that writes the values of the object’s instance variables to the typed stream. (Note that **NXWriteObject()** is equivalent to using **NXWriteType()** or **NXWriteTypes()** and specifying “@” in the format string; in the following discussion, **NXWriteObject()** will be used as a proxy for all these equivalent methods of writing objects.)

NXWriteRootObject() and **NXWriteObject()** differ in how they expect the object’s **write:** method to handle its **id** instance variables. **NXWriteObject()** expects to be able to archive every object referred to by **id** instance variables, as well as objects referred to by those objects, and so on. **NXWriteRootObject()** allows you to limit the scope of what’s archived by letting some **id** instance variables point to **nil** when they’re unarchived. The next sections describe how to set up a **write:** method and when to use these two functions.

A third function, **NXWriteRootObjectToBuffer()**, also begins the process of archiving a given object. This function doesn’t take a typed stream as an argument. Instead, it opens a typed stream on memory, writes the object to it, and returns a pointer to the memory buffer. This function is discussed in more detail below under “Opening and Closing a Typed Stream.”

The write: Method

Any Application Kit class or Common Class that declares instance variables already has a **write:** method that archives those instance variables. You need to supply a **write:** method for any class you create that adds instance variables. However, not every single instance variable needs to be archived. If an instance variable can be initialized by using the values of other instance variables, you don’t need to archive its value.

Note that **write:** messages shouldn’t be sent directly to objects. They should only be generated by the functions **NXWriteRootObject()** and **NXWriteObject()**.

Every **write:** method should begin with a message to **super**:

```
- write:typedStream {
    [super write:typedStream];
    . . . /* code for writing instance variables declared in this
           class */
}
```

This ensures that the object's class hierarchy and its inherited instance variables are archived. The body of the **write:** method uses the appropriate functions to archive the instance variables declared in that class. You can use any of the functions listed above in the "Archiving a Data Structure" section. If the object being archived has **id** instance variables, they're archived as described below.

Archiving id Instance Variables

An object's **id** instance variables can be archived in one of two ways, depending on whether the object referred to by the instance variable is an intrinsic part of the object being archived. If it is intrinsic, use **NXWriteObject()**, **NXWriteType()**, or **NXWriteTypes()**, which are all equivalent. If it's not intrinsic, use **NXWriteObjectReference()**. The following paragraphs explain the differences among these functions.

An object's **id** instance variables may contain inherent properties of the object to which they belong, or they might be necessary for the object to be usable. For example, a View's subview list is an intrinsic part of that View, just as a ButtonCell is needed for a Button to work properly. These kinds of instance variables are archived using **NXWriteObject()**. The following shows part of a View's **write:** method:

```
- write:(NXTypedStream *) typedStream {
    [super write:typedStream];
    NXWriteObject(typedStream, subviews);
    . . . /* code for writing other instance variables */
}
```

If you design a subclass of View that defines instance variables, you'll need to create a **write:** method that archives those instance variables. Since your method will begin with a message to **super**, the subviews list will be archived along with the View. Button objects don't define instance variables, so they inherit Control's **write:** method, which archives the **cell** instance variable.

In some cases, an object's **id** instance variables refer to other objects that act at the discretion of the object, such as its target or delegate, or that aren't inherently part of the object. A View's **Superview** and **window** instance variables aren't considered intrinsic to the View since you might want to hook up the View to another superview or to a different Window. These kinds of instance variables are archived using **NXWriteObjectReference()**.

NXWriteObjectReference() specifies that a pointer to **nil** should be written for the **id** passed in unless that object is an intrinsic part of some member of the data structure being archived. If the object is intrinsic, it will be archived and the pointer will point to the archived object.

Archiving an Object with id Instance Variables

When an object that includes any calls to **NXWriteObjectReference()** is archived, **NXWriteRootObject()** must be used to archive the object instead of **NXWriteObject()**. If the object being archived is based on the Application Kit, **NXWriteRootObject()** should be used since several Application Kit classes use **NXWriteObjectReference()**. Using **NXWriteRootObject()** will always give the desired result whether **NXWriteObjectReference()** is called or not. However, **NXWriteObject()** will raise an exception if used to archive an object that calls **NXWriteObjectReference()**.

NXWriteRootObject() makes two passes through the data structure being written. The first time, it defines the limits of the data to be written by including instance variables intrinsic to the data structure and by making a note of which instance variables are written with **NXWriteObjectReference()**. On the second pass, **NXWriteRootObject()** archives the data structure. Because of this two-pass implementation, **write:** methods are performed twice; therefore, **write:** methods shouldn't contain any code that has side effects.

As an example, consider a View that has a Button as one subview and a TextField, which is the target of the Button, as another subview. If you archive the Button, its ButtonCell will be written. The archived ButtonCell's **target** instance variable will point to **nil**. If you archive the View, however, the Button and the TextField will be archived since they're subviews. The ButtonCell will be archived since it's needed by the Button. The ButtonCell's **target** instance variable will point to the TextField since it's an intrinsic part of the View.

Unarchiving a Data Structure

The functions for unarchiving data are similar to the functions for writing:

Function	Data Type
NXReadType()	Single specified type
NXReadTypes()	Multiple specified types
NXReadArray()	An array
NXReadPoint()	An NXPoint structure
NXReadSize()	An NXSize structure
NXReadRect()	An NXRect structure
NXReadObject()	An id

With the exception of **NXReadObject()**, these functions take the same arguments as their counterparts for archiving. Rather than writing the data pointed to by the arguments,

however, the unarchiving functions read data from the typed stream into locations specified by the function’s arguments. **NXReadObject()** takes only a typed stream as an argument and returns the unarchived object’s **id**. The section “Unarchiving Objects” below contains more information about reading an object from a typed stream and initializing it.

In the following example, a **float**, an **int**, and a **char *** are read from a typed stream and stored in the locations specified by the last three arguments to **NXReadTypes()**.

```
float aFloat;
int anInt;
char *aCharStar;

NXReadTypes (typedStream, "fi*", &aFloat, &anInt, &aCharStar);
```

All the functions for reading check the type of data on the stream and raise an exception if the type isn’t what’s expected.

Unarchiving Objects

Unarchiving an object from a typed stream is initiated by a call to **NXReadObject()**. Because an object’s class hierarchy is archived with the object, **NXReadObject()** can determine the object’s class and allocate enough memory for a new instance of that class. It then initializes the object’s instance variables by sending it a **read:** message, which reads values for the instance variables from the typed stream.

The read: Method

read: methods have already been defined for all Application Kit classes and common classes that declare instance variables. You need to supply a **read:** method for any class you create that adds instance variables. As with **write:** methods, **read:** messages shouldn’t be sent directly to objects. They should only be generated by **NXReadObject()**.

Every **read:** method should begin with a message to **super:**

```
- read:typedStream {
    [super read:typedStream];
    . . . /* code for reading instance variables declared in this
           class */
}
```

This ensures that the object’s inherited instance variables are unarchived. The body of the **read:** method uses the appropriate functions to unarchive the instance variables declared in that class, in the order in which they were archived in the **write:** method. Any of the functions listed above in the “Unarchiving a Data Structure” section can be used to read values for instance variables.

Unarchived **id** instance variables are initialized to point either to an object or to **nil**, depending on whether the referenced object was archived. If **NXWriteObjectReference()** was used for the **id** instance variable and if the referenced object isn't an intrinsic part of any member of the structure that was archived, then the instance variable will point to **nil**. Otherwise, it'll point to the object. See “Archiving Objects” earlier for more information.

Values for other instance variables may not have been archived because they can be derived from others. Values for these instance variables should be computed in the **read:** method. Other initialization needed can be performed as described in the next section.

If you create a class, archive an instance of it, and later create a new version of that class (for example, you decide to add an instance variable), you can set up your **read:** method to read both versions. When a class is created, its version should be set using Object's **setVersion:** method:

```
@implementation MyClass:MySuperClass
+ initialize
{
    [MyClass setVersion:MYCLASS_CURRENT_VERSION];
    return self;
}
```

The **read:** method for this class can then check the version being unarchived by using **NXTypedStreamClassVersion()** and, if necessary, use different code for reading an old version:

```
- read:(NXTypedStream *)typedStream
{
    [super read:typedStream];
    if (NXTypedStreamClassVersion(typedStream, "MyClass") ==
        [MyClass version] {
        /* read code for current version */
        .
        .
    }
    else {
        /* read code for old version */
        .
        .
    }
}
```

Initializing an Object

Immediately after an object has been read from a typed stream, **NXReadObject()** sends it an **awake** message. This gives the object a chance to perform initialization tasks that can't be done in the **read:** method—that is, those tasks that require the entire object to be unarchived and in a usable state. For example, Window's **awake** method has the Window Server redisplay the window and assign it a window number. If you override any of the Application Kit's **awake** methods, your version should begin by sending an **awake** message to **super**.

After sending an **awake** message, **NXWriteObject()** sends the object a **finishUnarchiving** message. The purpose of this method is to allow you to replace the just-unarchived object with another one. If you implement a **finishUnarchiving** method, it should free the unarchived object and return the replacement object.

Opening and Closing a Typed Stream

The functions for archiving and unarchiving take an already opened typed stream as an argument. You can use one of three functions to open a typed stream, depending on whether you're archiving to or unarchiving from a file, memory, or some other destination or source:

- **NXOpenTypedStreamForFile()** returns a pointer to a typed stream opened on a specified file.
- **NXWriteRootObjectToBuffer()** opens a typed stream on memory and writes the given object to it using **NXWriteRootObject()**. The corresponding function for unarchiving, **NXReadObjectFromBuffer()**, opens a memory stream and reads an object from it.
- **NXOpenTypedStream()** takes an already opened NXStream structure as an argument and returns a pointer to a typed stream.

Regardless of what the typed stream is opened on, the same archiving code (or unarchiving code) can be used for a given data structure.

A Typed Stream on a File

NXOpenTypedStreamForFile()'s two arguments are the pathname of a file and a constant that indicates whether you'll be archiving or unarchiving:

```
NXTypedStream *typedStream;  
typedStream = NXOpenTypedStreamForFile("yourPathname", NX_WRITEONLY);
```

This function returns a pointer to a typed stream on memory and makes note of the fact that the stream is associated with a file. If you open the stream for archiving—by specifying **NX_WRITEONLY**—you can use any of the functions described above in “Archiving a Data Structure.” When you’ve finished, call **NXCloseTypedStream()**. This function saves the contents of the typed stream in the file, creating it if necessary, and closes the stream.

If **NX_READONLY** is specified, the typed stream is initialized with the contents of the file specified. You unarchive data by using any of the functions described above in “Unarchiving a Data Structure.” Then call **NXCloseTypedStream()** to close the typed stream.

A Typed Stream on Memory

NXWriteRootObjectToBuffer() and **NXReadObjectFromBuffer()** both open a stream on memory. They're particularly useful for archiving an object, writing it to the pasteboard, and then unarchiving it from the pasteboard. See "The Pasteboard" later in this chapter for an example of using these functions in conjunction with the pasteboard.

NXWriteRootObjectToBuffer() opens a memory stream, writes the object given as its argument by calling **NXWriteRootObject()**, and then closes the stream. It returns the size of the object written, in the location specified by the second argument, and a pointer to the memory buffer.

```
char *data;
int length;

data = NXWriteRootObjectToBuffer(anId, &length);
```

NXReadObjectFromBuffer()'s two arguments should be taken from a previous call to **NXWriteRootObjectToBuffer()**:

```
id someId;
someId = NXReadObjectFromBuffer(data, length);
```

NXReadObjectFromBuffer() calls **NXReadObject()** to read the object and then closes the typed stream and returns the object's **id**.

When you finish with the memory buffer, after a call to either **NXWriteRootObjectToBuffer()** or **NXReadObjectFromBuffer()**, free the buffer by calling **NXFreeObjectBuffer()**. This function takes the same arguments as **NXReadObjectFromBuffer()**.

Using an NXStream Structure

In addition to the functions described above for opening a typed stream on file or memory, you can open a typed stream by passing **NXOpenTypedStream()** a pointer to an NXStream structure. NXStream structures can be opened on Mach ports, memory, files, or even objects. To obtain an NXStream pointer, use the functions described in "Streams" earlier in this chapter.

NXOpenTypedStream()'s second argument should be **NX_READONLY** or **NX_WRITEONLY** to specify whether you'll be archiving or unarchiving. This constant should be the same as that used to open the NXStream structure. When you finish archiving or unarchiving, you need to close the NXStream structure and the typed stream. The section on "Streams" describes how to close NXStream structures. Use **NXCloseTypedStream()** to close the typed stream.

The Defaults System

Through the defaults system, you can allow users to customize your application to match their preferences. For example, you can let users express a preference for where the main menu of your application should come up the next time it's launched. This preference will override the default location of the main menu that users can set with the Preferences application. An application records such preferences by assigning default values to a set of parameters. Each user has a defaults database named **.NextDefaults**, which resides in the **.NeXT** subdirectory in the user's home directory, for storing these default values.

Warning: The **.NeXTDefaults** file should never be accessed directly. Values in it can be read and written using the functions and commands described in this section.

Since the defaults database is a system resource, it isn't owned by any single application. In fact, any application can store values for parameters or get values stored by another application. For example, the ChoosePrinter panel writes to the defaults database to store the name of the printer selected by the user. Another application may want to obtain this printer specification from the database. Applications can use the functions discussed below to read values from and write them to the database. These functions are described in detail in *NeXTstep Reference, Volume 2*.

Creating a Registration Table

The registration table allows an application to efficiently read default values for a set of parameters without having to open and close the **.NextDefaults** database to obtain each value. The table consists of a list of pairs; each pair is composed of a parameter name and a corresponding default value. The registration table is created at run time by opening the database once to read default values for the parameters the application will use. Every application should create its registration table early in the program, before any default values are needed.

To create this table, call **NXRegisterDefaults()** and give it two arguments: a character string specifying the name of an application, or owner, and an **NXDefaultsVector** structure. Like the registration table, this structure consists of a list of pairs of parameter names and default values. (It's defined in the header file **appkit/defaults.h**.)

The **NXDefaultsVector** structure serves two purposes. First, it provides a complete list of all parameters that the application will use. Values for all the parameters specified are placed in the registration table at once, so the database doesn't need to be opened and closed for subsequent uses of the parameters. (However, if the application later asks for values for parameters that aren't registered, the database will be opened, read, and closed again.) Second, the structure allows the programmer to suggest values for the parameters. These values are used if the user hasn't stated a preference for a specific value.

A good place to call **NXRegisterDefaults()** is in the **initialize** method of the class that will use the parameters. The following example registers the values in **WriteNowDefaults** for the owner **WriteNow**:

```
+ initialize
{
    static NXDefaultsVector WriteNowDefaults = {
        {"NXFont", "Helvetica"},
        {"NXFontSize", "12.0"},
        {NULL}
    };

    NXRegisterDefaults("WriteNow", WriteNowDefaults);

    return self;
}
```

NXRegisterDefaults() creates a registration table that contains a value for each of the parameters listed in the **NXDefaultsVector** structure. (Note that **NULL** is used to signal the end of the **NXDefaultsVector** structure.) This value will be the one listed in the structure if there's no value for that parameter in the database, as described below.

A user's database may contain values for parameters stored multiple times, each with a different owner. For example, the **NXFont** parameter can have the value **Ohlfs** with a **GLOBAL** owner, **Times** for the owner **WriteNow**, and **Courier** for the owner **Mail**. When searching a user's database for the parameters listed in the **NXDefaultsVector** structure, **NXRegisterDefaults()** ignores values owned by an application different from the one used as its argument. If it finds a parameter and owner that matches those passed to it as arguments, the corresponding value from the user's database rather than the value from the **NXDefaultsVector** structure is placed in the registration table. If no parameter-owner match is found, **NXRegisterDefaults()** searches the database's global parameters—that is, those owned by **GLOBAL**—for a match, and, if it finds one, places the corresponding value in the registration table. (Global parameters are discussed in a later section.) If a parameter isn't found in the user's database, the parameter-value pair listed in the **NXDefaultsVector** structure is placed in the registration table.

Note: When creating their own parameters, applications should use the full market name of their product as the owner of the parameter to avoid colliding with already existing parameters. Noncommercial applications might use the name of the program and the author or institution.

If the application was launched from the command line, any parameter values specified there will be used, overriding values listed in the database and the **NXDefaultsVector** structure. See “The Command Line” below for more information.

To summarize, this is the precedence ordering used to obtain a value for a given parameter for the registration table:

1. The command line
2. The defaults database, with a matching owner
3. The defaults database, with the owner listed as GLOBAL
4. The NXDefaultsVector structure passed to **NXRegisterDefaults()**

Reading Default Values

To get a value for a parameter, you typically call **NXGetValue()**. This function takes an owner and a parameter as arguments, as shown below, and returns a **char** pointer to the default value for that parameter.

```
char *myDefaultFont;  
myDefaultFont = NXGetValue("WriteNow", "NXFont");
```

NXRegisterDefaults() should already have been called, so **NXGetValue()** first looks in the registration table, where usually it will find a matching parameter and value. If **NXGetValue()** doesn't find a match in the registration table (which would only be the case if you hadn't listed all parameters when you called **NXRegisterDefaults()**), it searches the **.NextDefaults** database for the owner and parameter. If still no match is found, it searches for a matching global parameter, first in the registration table and then in the database. If the value is found in the database rather than the table, **NXRegisterDefaults()** registers that value for subsequent use.

Occasionally, you may want to search only the database for a default value and ignore the command line and the registration table. For example, you might want a value that another application may have changed after the table was created. In these rare cases, call **NXReadDefault()**, which takes an owner and the parameter as arguments and looks in the database for an exact match. It doesn't look for a global parameter unless GLOBAL is specified as the owner. If a match is found, a **char** pointer to the default value is returned; if no value is found, NULL is returned.

After obtaining a value from the database with **NXReadDefault()**, you may want to write it into the registration table with **NXSetDefault()**, which is described below.

Writing Default Values

If you have allowed a user to customize an application, you probably want to write new values into the user's **.NextDefaults** database to store these preferences. You probably also want to put the values in the registration table for efficient access by **NXGetValue()**. In addition, at various points in your program, you may want to

update the registration table with any recent changes to the database. The following paragraphs explain the functions that manipulate the contents of the database and the registration table.

NXWriteDefault() writes a default value into both the database and the registration table. It takes an owner, a parameter, and a default value for that parameter as arguments:

```
NXWriteDefault("WriteNow", "NXFont", "Helvetica");
```

In this example, the NXFont parameter and its value Helvetica are written into both the database and the registration table for the owner **WriteNow**.

Similarly, **NXWriteDefaults()** writes a vector of default values for the given owner into the database and registers them.

```
static NXDefaultsVector WriteNowDefaults = {  
    {"NXFont", "Times"},  
    {"NXFontSize", "12.0"},  
    {NULL},  
};  
NXWriteDefaults("WriteNow", WriteNowDefaults);
```

Both **NXWriteDefault()** and **NXWriteDefaults()** return the number of successfully written values. To maximize efficiency, you should use one call to **NXWriteDefaults()** rather than several calls to **NXWriteDefault()** to write multiple values. This will save the time required to open and close the database each time a value is written.

NXSetDefault() takes an owner, a parameter, and a value for that parameter as arguments:

```
NXSetDefault("WriteNow", "NXFont", "Helvetica");
```

The parameter and its default value are placed in the registration table, but they aren't written into the **.NextDefaults** database.

Since other applications can write to the database, at various points the database and the registration table might not agree on the value of a given parameter. (The user can also write to the database, as described in the next section.) You can update the registration table with any changes that have been made to the database since the table was created by calling **NXUpdateDefault()** or **NXUpdateDefaults()**. Both functions compare the table and the database. If a value is found in the database that is newer than the corresponding value in the registration table, the new value is written into the registration table.

NXUpdateDefault() updates the value for the single parameter and owner given as its arguments:

```
NXUpdateDefault("WriteNow", "NXFont");
```

NXUpdateDefaults(), which takes no arguments, updates the entire registration table. It checks every parameter in the registration table, determines whether a newer value exists in the database, and puts any newer values it finds in the registration table.

NXRemoveDefault() removes a specified parameter for the given owner from the **.NextDefaults** database.

```
NXRemoveDefault("WriteNow", "NXFontSize");
```

Changing the Defaults Database from a Shell Window

In addition to the functions described above, the following three commands can be used in a Terminal or Shell window to read and write default values:

- **dread -l** reads all the values in the defaults database and sends them to **stdout**. Instead of the **-l** option, you can specify a particular owner and a parameter; if no owner is specified, it's assumed to be **GLOBAL**.
- **dwrite** takes an owner, a parameter, and a value as arguments and writes the value into the defaults database. If the **-g** option is used, the owner is assumed to be **GLOBAL**. If no arguments are given, input is taken from **stdin**.
- **dremove** removes the parameter named as an argument from the database. If an owner is specified as the first argument, **dremove** removes that owner's parameter-value pair; if the **-g** option is used, the owner is assumed to be **GLOBAL**. If no arguments are given, input is taken from **stdin**.

All arguments for these commands should be separated by spaces. For more information on using these commands, see their UNIX manual pages.

The Command Line

Without changing the **.NextDefaults** database, you can temporarily override values in the database or supply values for parameters that don't exist in the database. To do this, specify the desired values when launching an application from a Shell or Terminal window, as shown below:

```
Edit -WidthInChars 100 -HeightInChars 120 SomeFile.m &
```

In this example, Edit will be launched, in a window that's 100 characters wide and 120 characters high. When **NXRegisterDefaults()** is called, the command-line values will be placed in the registration table, overriding values specified by the database and the **NXDefaultsVector** structure. However, these values will not be written into the database.

System and Global Parameters

The Application Kit registers values for system and global parameters. System parameters are used by all applications for such things as determining which printer to use and which font to use in attention panels. Values for system parameters should remain constant across the system, so applications shouldn't overwrite system values. Values for global parameters are used by applications if there's no application-specific value. Global parameters determine the location of the main menu and the font used to display text, for example. Applications are encouraged to declare their own, application-specific, values for global parameters.

The following sections list the system and global parameters and describe their meaning. Parameters owned by the Workspace Manager are also discussed since they sometimes affect multiple applications. The parameters owned by Edit, Shell, and Terminal are described in the *NeXT Development Tools* manual. (All parameter names and their values are character strings; for simplicity, they're shown below without quotation marks.)

System Parameters

Applications obtain values for system parameters by specifying "System" as the owner in one of the functions described above for reading default values. Users can set values for some of these parameters through the Preferences applications. (For more information about Preferences, see *The NeXT User's Reference Manual*.)

The system parameters and their initially registered values are listed below.

Parameter	Initial Value
SystemAlert	Both
UnixExpert	NO
PublicWindowServer	NO
Umask	18
BrowserSpeed	50
Printer	Local_Printer
PrinterHost	NULL
PrinterResolution	400
SystemFont	Helvetica
BoldSystemFont	Helvetica-Bold
ScrollerButtonDelay	0.5
ScrollerButtonPeriod	0.025

Users can set default values for the first five parameters through the Preferences application.

- The SystemAlert parameter allows applications to offer voice as well as panels for system alerts.

- If the UnixExpert parameter is set to NO, all UNIX system files will be hidden.
- The PublicWindowServer parameter determines whether processes that are not descended from the Workspace Manager have host access to the computer.
- Values for the Umask parameter are integers that correspond to the octal values used by the **umask()** system call to set the file-creation mask.
- The BrowserSpeed parameter determines how fast scrolling will be when the user clicks on a browser's scroll button. Values for this parameter can range from 0 to 100.

The next three parameters concern printing.

- Printer specifies the printer that will be used. Valid printers are listed in the ChoosePrinter panel that's opened through the Choose button in the Print panel.
- PrinterHost specifies the host machine of the printer. The default value, NULL, indicates the local printer's host.
- Printing can be performed with a resolution of either 300 or 400 dpi.

The next two pairs of parameters specify the font used by the system to display text and how scroll buttons will respond.

- The parameters SystemFont and BoldSystemFont are stored in global variables NXSystemFont and NXBoldSystemFont, respectively, for easy use by the Application Kit. The Kit uses these variables, for example, to display text in attention panels and in Cells of type NX_TEXTCELL.
- The value for the ScrollerButtonDelay parameter specifies how many seconds the user must hold down the mouse button to make a scroll button repeat. ScrollerButtonPeriod indicates the interval, also in seconds, at which the scrolling action will be repeated if the user continues to hold down the mouse button.

Global Parameters

The Application Kit registers values for global parameters in the Application object's **initialize** method. Users can set values for some of these parameters using the Preferences application. Preferences writes values specified by a user into the defaults database (with owner set to GLOBAL) so they will override the initial values supplied by the Application object.

The global parameters that have been defined on the NeXT computer and their initially registered values are listed below. The following paragraphs discuss the possible values for these parameters and their meaning.

Parameter	Initial Value
NXFont	Helvetica
NXFontSize	12
NXMenuX	-1.0
NXMenuY	1000000.0
NXFixedPitchFont	Ohlfs
NXFixedPitchFontSize	10
NXPaperType	Letter
NXMargins	72 72 90 90
NXAutoLaunch	NO
NXCaseSensitiveBrowser	NULL
NXHost	NULL
NXOpen	NULL
NXOpenTemp	NULL
NXShowAllWindows	NULL
NXShowPS	NULL
NXMallocDebug	1
NXPSName	NULL

Users can set default values for the first four parameters through the Preferences application.

- Because the initial value for the NXFont parameter is 12-point Helvetica, applications that create documents will use this font by default. However, many applications assign their own values to the NXFont and NXFontSize parameters, and most of these applications also provide a Font panel through which users can change the values.
- NXMenuX and NXMenuY specify the location of the main menu of the application. The initially registered values are off the screen, so applications will probably want to supply their own values.

The next two parameters affect the font of applications that use fixed-width fonts, such as Shell and Terminal.

- The default value is 10-point Ohlfs font. NXFixedPitchFont must be set to a fixed-width font, such as Courier or Ohlfs, rather than a variable-width font, such as Times.

The next pair of parameters concerns printing.

- NXPaperType must be one of the standard paper types for PostScript documents such as Letter, Legal, or A4.

- The NXMargins parameter specifies the printing area on the page; the initial setting is appropriate for letter-size paper.

Values for the next two parameters indicate whether an application was automatically launched at login and whether an application's browser ignores case.

- The Workspace Manager passes YES as the value for NXAutoLaunch if the application was automatically launched when the user logged in. (See the description of the LaunchThese parameter below under "Workspace Manager's Parameters.")
- Applications that create browsers can use the NXCaseSensitiveBrowser parameter to determine whether they should ignore case when alphabetizing the browser's contents.

You can use the command line to specify values for the last six parameters, which are used only at launch time and shouldn't be written to the database. See "The Command Line" above for more information on how to do this.

- The NXHost parameter enables you to run an application on one machine while sending the PostScript code generated to another machine. The host machine will display windows and accept events from the user.
- The NXOpen parameter specifies the name of the file to be opened by the application being launched. If NXOpenTemp is used to specify a file, that file won't be saved when the application quits unless you explicitly tell the application to save it.

NXShowPS, NXShowAllWindows, and NXMallocDebug control the display of debugging output. By default, NXShowPS and NXShowAllWindows are turned off.

- NXShowPS writes to **stderr** both the PostScript code produced by the application and values returned from the PostScript interpreter to the application.
- NXShowAllWindows displays all off-screen windows created by the application. These windows typically contain Bitmap objects used for compositing into on-screen windows.
- The value of NXMallocDebug is passed to the **malloc_debug()** function, which controls the amount of error checking that **malloc()** performs. The UNIX manual page on **malloc()** contains more information about both these functions.

NXPSName is used to establish a connection with the Window Server:

- The Application Kit uses the value of NXPSName to look up the Window Server from the Network Name Server.

Workspace Manager's Parameters

The parameters belonging to the Workspace Manager and their initial values are shown below.

Parameter	Initial Value
LaunchThese	Preferences
IconsSnapTo	YES
BrowserColWidth	120
ApplicationPaths	~/Apps:/LocalApps:/NextApps:/NextDeveloper/Apps: /NextAdmin:/NextDeveloper/Demos
CoreLimit	NULL
BrowserX	265
BrowserY	287
BrowserW	534
BrowserH	296

The first two parameters can be set through panels brought up by the Workspace Manager. They're documented in more detail in *The NeXT User's Reference Manual*.

- LaunchThese indicates which applications are automatically launched when the user enters the workspace.
- IconsSnapTo specifies whether icons displayed in the Icon view are aligned on the grid.

The next three parameters can be set by using the command line at launch time or by writing them into the database from a Shell or Terminal window.

- BrowserColWidth specifies the width of the columns of the Directory Browser.
- The Workspace Manager searches for application programs in the colon-separated directory list in ApplicationPaths. See “Paths” in Chapter 2, “The NeXT User Interface,” for more information about how the Workspace Manager uses this parameter.
- CoreLimit places a limit on the size of core files for Workspace Manager and its children. If a program dies, the system will write a core file if you have write permission in the working directory of the program and if CoreLimit is larger than the size of the core image. The default value, NULL, means that Workspace Manager inherits its parent's core limit.

The remaining four parameters set the position and size of the Browser.

- BrowserX and BrowserY specify the x- and y-coordinates of the lower left corner of the Directory Browser window. BrowserW and BrowserH specify its width and height. Values for these parameters shouldn't be set directly; they're set simply by moving or resizing the Browser on-screen.

The Pasteboard

The pasteboard is the principal means by which users can move data within and between applications. It supports a cut/copy/paste user-interface paradigm. To the user, there is a single pasteboard that all applications share, providing a unified environment. Also from the user's point of view, there is a single thing in the pasteboard at a given time. Internally, however, the pasteboard may contain more than one representation of its contents. For example, if a user cuts a piece of text from a word processor, that text replaces whatever was previously held in the pasteboard; however, that text may be represented in the Pasteboard object by an ASCII string and a piece of PostScript code at the same time.

Using the Pasteboard

Applications using the pasteboard perform all operations through a single instance of the Pasteboard class. This global Pasteboard object is accessed by sending a **pasteboard** message to the Application object:

```
id myPboard;
myPboard = [NXApp pasteboard];
```

The Pasteboard object manages all communications with **pbs**, the pasteboard server. All data read from or written to the pasteboard goes through **pbs**. Data for a particular type is transmitted as a single, contiguous buffer of memory. Since data is transmitted using Mach messaging, these buffers are shared among applications, making the communication very efficient for large quantities of data. Essentially, each application that has used the data has a pointer to the same, shared physical memory, even though it may appear in different ranges of their address spaces.

Declaring Data Types

When an application performs a copy (or a cut), it first becomes the owner of the pasteboard by declaring what types of data it will put in the pasteboard. It does this with the **declareTypes:num:owner:** method. The first two arguments for this method are a list of all possible representations for the selection being copied and the number of types in that list. An application can write any of the standard pasteboard data types defined by NeXT. It can also write its own data types for its own use, or for use among a cooperating set of applications. (Data types are named by null-terminated character strings.) The standard data types and their corresponding global variables, which are declared in the Application Kit header file **Pasteboard.h**, are listed below.

Data Type	Global Variable
Plain ASCII text	NXAsciiPboard
Rich Text Format (RTF) version 1.0	NXRTPboard
Encapsulated PostScript (EPS) version 1.2	NXPostScriptPboard
Tag Image File Format (TIFF) version 5.0	NXTIFFPboard
NeXT sound pasteboard data type	NXSoundPboard

The ASCII and RTF types both describe text. Clients of the pasteboard that handle text should always declare and be able to accept ASCII data. If they can also produce or read RTF, they should declare that type as well.

The pasteboard owner, which is the third argument for **declareTypes:num:owner:**, promises to supply data in all the representations declared. When copying data to the pasteboard, as described below, the owner can choose to delay writing a type until that type is requested, or it can supply all representations at one time. If writing will be delayed, the owner must be an object that won't be freed so that it can be informed when data has been requested. If all representations will be supplied at one time, the owner can be NULL.

Copying Data to and Reading it from the Pasteboard

After declaring the data types, data can be written to the pasteboard with the **writeType:data:length:** method. The first argument specifies the type of the data, and the second points to the data to be written. The length argument specifies the number of bytes of data. This method is called each time a different type is written to the pasteboard.

When an application performs a paste, it first examines the available data types in the pasteboard. The **types** method returns a null-terminated array of character strings describing the available types. If the application finds a data type that's appropriate, it requests the data with the **readType:data:length:** method. If that data representation has not yet been written to the pasteboard, the owner specified in the **declareTypes:num:owner:** method is sent a **provideData:** message with the type requested as an argument. The owner must then write that type of data to the pasteboard. (If the application quits before supplying all declared data types, a **provideData:** message will also be sent. This only works if the application quits using Application's **terminate:** method.)

Applications that do significant calculation to import a certain type may be able to save this work on repeated pastes of the same data by checking the change count. The change count is an integer (returned by the **changeCount** method) that increments every time a set of types for the pasteboard is declared. If the change count is the same as it was during a previous paste, the same data is being imported.

The following section gives examples of the process of copying data to and reading it from the pasteboard. All functions mentioned below are described in more detail in *NeXTstep Reference, Volume 2*.

Examples of Preparing and Parsing Data

An NXStream or NXTypedStream structure can be helpful in creating and interpreting the buffers of data that the pasteboard deals with. They're similar to the stream model and interface in the UNIX **stdio** library, but they can read and write to memory and Mach ports as well as UNIX file descriptors. Using a stream, the same code can be used to interpret a buffer of a certain data type from the pasteboard as can be used to read a disk file of the same format. (See “Streams” in this chapter for more information.) A special kind of data stream, a typed stream, should be used for copying Objective-C objects to the pasteboard. Typed streams are discussed in more detail in “Archiving to a Typed Stream” earlier in this chapter.

The next two sections contain examples of using a stream and a typed stream with the pasteboard.

Using a Stream

In the following example, a View writes the PostScript representing itself to a stream and then copies it to the pasteboard.

```
- copy:sender
{
    id      pb = [NXApp pasteboard];
    NXStream *stream;
    char    *data;
    int     length;

    [pb declareTypes:&NXPostScriptPboard num:1 owner:self];
    stream = NXOpenMemory(NULL, 0, NX_WRITEONLY);
    [self copyPSCodeInside:NULL to:stream];
    NXGetMemoryBuffer(stream, &data, &length, &maxLength);
    [pb writeType:NXPostScriptPboard data:data length:length];
    NXCloseMemory(stream, NX_FREEBUFFER);
    return self;
}
```

The **declareTypes:num:owner:** method readies the pasteboard to receive a single type of data, PostScript. Then a stream that writes to memory is opened using **NXOpenMemory()**. Next the View’s PostScript code is written to the stream with the **copyPSCodeInside:to:** method. The contents of the stream are obtained with **NXGetMemoryBuffer()** and are then transferred to the pasteboard through **writeType:data:length:**. Finally, the stream is closed.

This is what the corresponding **paste:** method might look like:

```
- paste:sender
{
    id      pb = [NXApp pasteboard];
    char   **type;
    char   *data;
    int    length;
    NXStream *stream;

    for(type = [pb types]; *type; type++)
        if(!strcmp(*type, NXPostScriptPboard))
            break;

    if(*type) {
        [pb readType:NXPostScriptPboard data:&data length:&length];
        stream = NXOpenMemory(data, length, NX_READONLY);
        /* parse PostScript data using NXGetc() or NXScanf() */
        .
        .
        .NXCLOSEMEMORY(stream, NX_FREEBUFFER);
    }
    else
        /* invalid data type - raise an exception */;
    .
    .
    return self;
}
```

The **paste:** method first ensures that PostScript code is one of the pasteboard's available data types. Then it reads the PostScript data from the pasteboard with **readType:data:length:** and opens a memory stream on the data using **NXOpenMemory()**. The data can be parsed using **NXGetc()** or **NXScanf()** and pasted in, after which the stream is closed. Data read from the pasteboard is allocated using **vm_allocate()**, so it must be freed using **vm_deallocate()**. **NXCLOSEMEMORY()** does this automatically if **NX_FREEBUFFER** is specified.

Using a Typed Stream

A typed stream should be used to copy an Objective-C object to and read it from the pasteboard. A typed stream writes an object's class hierarchy as well as both the data type and value of the object's instance variables.

The example below writes an object to a typed stream using the function **NXWriteRootObjectToBuffer()** and then puts it on the pasteboard.

```

-copy:sender
{
    const char *const types[1] = {"PrivateTypes"};
    id         pb = [NXApp pasteboard];
    char      *data;
    int        length;

    [pb declareTypes:types num:1 owner:self];
    data = NXWriteRootObjectToBuffer(SelectionList, &length);
    [pb writeType:types[0] data:data length:length];
    NXFreeObjectBuffer(data, length);
    return self;
}

```

In this example, the data to be written to the pasteboard exists in SelectionList, which might be a List object, for example. **NXWriteRootObjectToBuffer()** opens a typed stream on memory, writes the object given as its argument, and then closes the stream. It also returns both the size of the object (in the location specified by **length**) and a pointer to the memory buffer itself, which is truncated to the size of the object. The contents of this buffer can then be written to the pasteboard with **writeType:data:length:**. Finally, the typed stream and the data are freed with **NXFreeObjectBuffer()**.

The following method reads the object from the pasteboard:

```

-paste:sender
{
    char **type;
    id      pb = [NXApp pasteboard];
    char   *data;
    int     length;
    id      PasteList;

    for(type = [pb types];*type;type++) {
        if(!strcmp(*type,"PrivateTypes"))
            break;

    }
    if(*type) {
        [pb readType:*type data:&data length:&length];
        pasteList = NXReadObjectFromBuffer(data, length);
        NXFreeObjectBuffer(data, length);
    }
    /*code for pasting in the data*/
    . .
    return self;
}

```

The **for** loop shown above checks whether the desired data type is in the pasteboard. If so, the corresponding data is read from the pasteboard into the typed stream with the **readType:data:length:** method. **NXReadObjectFromBuffer()** then opens a typed stream, reads the data into **data**, closes the stream, and returns the buffer. Since in this case the buffer won't be reread, it's freed with **NXFreeObjectBuffer()**.

Responding to Cut, Copy, and Paste

Interface Builder provides your application with a main menu containing a standard Edit submenu with Cut, Copy, and Paste commands. These commands are initialized to send the **cut:**, **copy:**, and **paste:** messages to the first responder, and thus through the responder chain. Editable Application Kit classes, like Text, implement the **cut:**, **copy:**, and **paste:** methods, and therefore respond to these menu choices without any explicit connections from the menu items. An application's View subclasses that support cut, copy, and paste should allow themselves to become the first responder, implement **cut:**, **copy:**, and **paste:** methods, and let the standard menu items and the responder chain find these implementations.

Exception Handling

An exceptional condition is one that interrupts the normal flow of program execution. Each application can interpret different types of conditions as exceptional. For example, one application might view as exceptional the attempt to save a file in a directory that's write-protected. In this sense, an exceptional condition can be equivalent to an error. Another application might interpret the user's keypress as an exceptional condition: an indication that a long-running process should be aborted.

A robust application must be able to respond appropriately to exceptional conditions. Depending on the context, this might mean:

- Terminating normal processing
- Freeing dynamically allocated memory
- Closing files
- Restarting execution at some other point in the program

Exceptional conditions can occur deep within a calling sequence—within a function that's called by another function that's called by yet another function, and so on. Responding to the condition might entail backing out of each of these functions in the reverse order that they were called, cleaning up as necessary at each level along the way. This process is known as “unwinding the call stack.”

Traditionally, exceptional conditions are announced through a function's return value. This system has several disadvantages. Only limited information about the condition can be coded in a single return value. For example, the UNIX manual page for **fopen()** states that the function returns NULL if it's unable to open a file, if too many files are already open, or if other needed resources can't be allocated. In addition, for the notification of the exceptional condition to propagate up the call stack, each function along the way must

check return values and respond appropriately. Failing this, information about the exceptional condition can be lost. Finally, providing for exceptional cases can obscure the normal pathways through your code:

```
if ((returnValue1 = function1()) == NULL) goto onError;
if ((returnValue2 = function2()) == NULL) goto onError;
if ((returnValue3 = function3()) == NULL) goto onError;

onError:
    /* Check where the error occurred and take remedial action */
```

A good exception handling system must provide the programmer with a unified and organized approach for responding to exceptional conditions once they've been identified at any level within an application. The following sections describe the system used in NeXTstep and available for use in applications built with NeXTstep.

Note: The exception handling system described here is distinct from the one used within the Mach operating system. See the *NeXT Operating System Software* manual for information on that system.

Detecting Exceptional Conditions

Before an exceptional condition can be responded to, it must be detected. Typically, an exceptional condition is discovered through the return value of a function or method, especially those that access data from the file system. An example is reading a file into memory, as this program excerpt illustrates:

```
NXStream *stream;
char     *theFile = "/me/filename", *buffer = NULL;
int      length, maxlenlength;

if ((stream = NXMapFile(theFile, NX_READONLY)) != NULL) {
    NXGetMemoryBuffer(stream, &buffer, &length, &maxlength);
    NXClose(stream);
} else {
    /* exceptional condition has been detected */
}
```

If **NXMapFile()** is unable to map the file into memory, it returns NULL, indicating an exceptional condition. Routines that write data generally have a similar system of reporting such conditions.

Even if the data can be read or written, it may not be usable to the program. Applications that run consistency checks on data may also use the exception handling system in case of data inconsistency.

Another situation where the exception handling system can be used is when the user wants to interrupt a long-running operation. In the following example, the application displays an attention panel to alert the user of its current state and to give the user the choice of aborting the operation:

```
id alert;
NXModalSession session;
int runState;
BOOL done;

alert = NXGetAlertPanel(NULL, "Doing something time-consuming.
                                Please wait . . . ", "Stop", NULL, NULL);
[NXApp beginModalSession: &session for:alert];
runState = NX_RUNCONTINUES;
done = NO;
while (!done && runState == NX_RUNCONTINUES) {
    runState = [NXApp runModalSession: &session];
    /* Do small portion of lengthy process. */
    /* Set done == YES when the process is finished. */
}

[NXApp endModalSession: &session];
[alert orderOut:self];
NXFreeAlertPanel(alert);
if (runState == NX_ALERTDEFAULT) {
    /* exceptional condition has been detected */
}
```

Program execution continues in the **while** loop either until the incremental processing finishes

```
done == YES
```

or until the user clicks the panel's Stop button:

```
runState != NX_RUNCONTINUES
```

If the user has interrupted processing, the statements in the body of the **if** construction are executed and the exception is detected. (See “Modal Sessions” in Chapter 7, “Program Dynamics,” for more information.)

Raising an Exception

Once an exceptional condition is detected, it must be propagated to the routine or routines that will handle it, a process referred to as “raising an exception.” In the NeXT exception handling system, exceptions are raised by calling the macro **NX_RAISE()**, as defined in the header file **streams/error.h**. This routine takes three arguments:

```
void NX_RAISE(int code, const void *data1, const void *data2)
```

The first, **code**, is an integer that identifies the exception. As described in the section “Exception Codes” below, some code ranges are reserved for the Application Kit, the Display PostScript client library, and other software modules; you can define codes for exceptional conditions that might occur in your application.

The second two arguments are pointers to arbitrary data about the exception. For example, if a function’s return value initiated the call to **NX_RAISE()**, you could use **data1** to pass the return value to the exception handler. Or, if the exception handler displays a panel in response to the exception, you could use **data2** to pass the text string to be displayed in the panel.

NX_RAISE() works by calling a function that’s registered as the exception raiser; for applications using the Application Kit, this function is **NXDefaultExceptionRaiser()**. **NXSetExceptionRaiser()** and **NXGetExceptionRaiser()** give you access to the exception raiser, although it’s unlikely that you’ll find a need to alter it.

Handling an Exception

Calling **NX_RAISE()** initiates the propagation of the exception and passes data about it. Where and how the exception is handled depends on where you make the call to **NX_RAISE()**. Let’s first look at a simple case.

In general, **NX_RAISE()** is called within the domain of an *exception handler*. An exception handler is a control structure created by the macros **NX_DURING**, **NX_HANDLER**, and **NX_ENDHANDLER**, as shown in Figure 8-1.

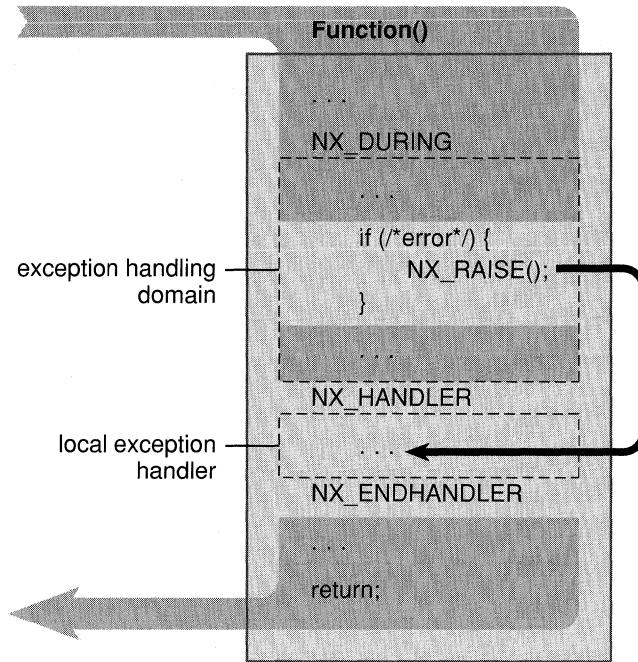


Figure 10-1. Flow of Control in an Exception Handler

The section of code between `NX_DURING` and `NX_HANDLER` is the *exception handling domain*; the section between `NX_HANDLER` and `NX_ENDHANDLER` is the *local exception handler*. The normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if `NX_RAISE()` is called. A call to `NX_RAISE()` causes program control to jump to the first executable line following `NX_HANDLER`, as indicated by the black arrow.

Although you can call `NX_RAISE()` directly within the exception handling domain, it's more often called indirectly within one of the procedures called from the domain. No matter how deeply in a call sequence the call to `NX_RAISE()` is made, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in the next section). In this way, exceptions raised at a low level can be caught at a high level.

Besides transferring execution to the local exception handler, a call to `NX_RAISE()` initializes the variable `NXLocalHandler` of type `NXHandler`, as defined in `streams/errors.h`. This variable is defined only within the local exception handler and contains those structure members that correspond to the arguments passed to `NX_RAISE()`: `NXLocalHandler.code`, `NXLocalHandler.data1`, and `NXLocalHandler.data2`. These members transfer information about the exception to the code within the local exception handler.

For example, in the following program excerpt, the local exception handler displays an attention panel after detecting an exception having the code **error_one** (see “Exception Codes” below for information on defining exception codes):

```
    . . .
NX_HANDLER
switch(NXLocalHandler.code) {
    case error_one:
        NXRunAlertPanel ("Error Panel",
                         NXLocalHandler.data1, "OK", NULL, NULL);
        break;
    case error_two:
        . . .
}
NX_ENDHANDLER
```

If an exception of type **error_one** is raised, an attention panel appears and displays the text referred to by **NXLocalHandler.data1**.

Calling **NX_RAISE()** is one way for program execution to leave the exception handling domain; three other ways are permitted:

- “Falling off the end”
- Calling **NX_VALRETURN()**
- Calling **NX_VOIDRETURN**

“Falling off the end” is simply the normal execution pathway introduced above. After all appropriate statements within the domain are executed (and no exception is raised), execution continues on the line following **NX_ENDHANDLER**. Alternatively, you can return control to the caller from within the domain by calling **NX_VALRETURN()** or **NX_VOIDRETURN**, depending on whether you need to return a value.

You can’t use **goto** or **return()** to exit an exception handling domain—errors will result. Nor can you use **setjmp()** and **longjmp()** if the jump entails crossing an **NX_DURING** statement. Since in many cases you won’t know if the NeXTstep code that your program calls has exception handling domains within it, it’s generally not recommended that you use **setjmp()** and **longjmp()** in your application.

If an exception is raised and execution begins within the local exception handler, it either continues until all appropriate statements are executed (falling off the end of the local exception handler), or the exception is raised again to invoke the services of an encompassing exception handler, as described in the next section.

Nested Exception Handlers

Exception handlers can be nested so that an exception raised in an inner domain can be treated by the local exception handler and any number of encompassing exception handlers. This hierarchy of exception handlers is accessed with the macro **NX_RERAISE**, as illustrated in Figure 8-2.

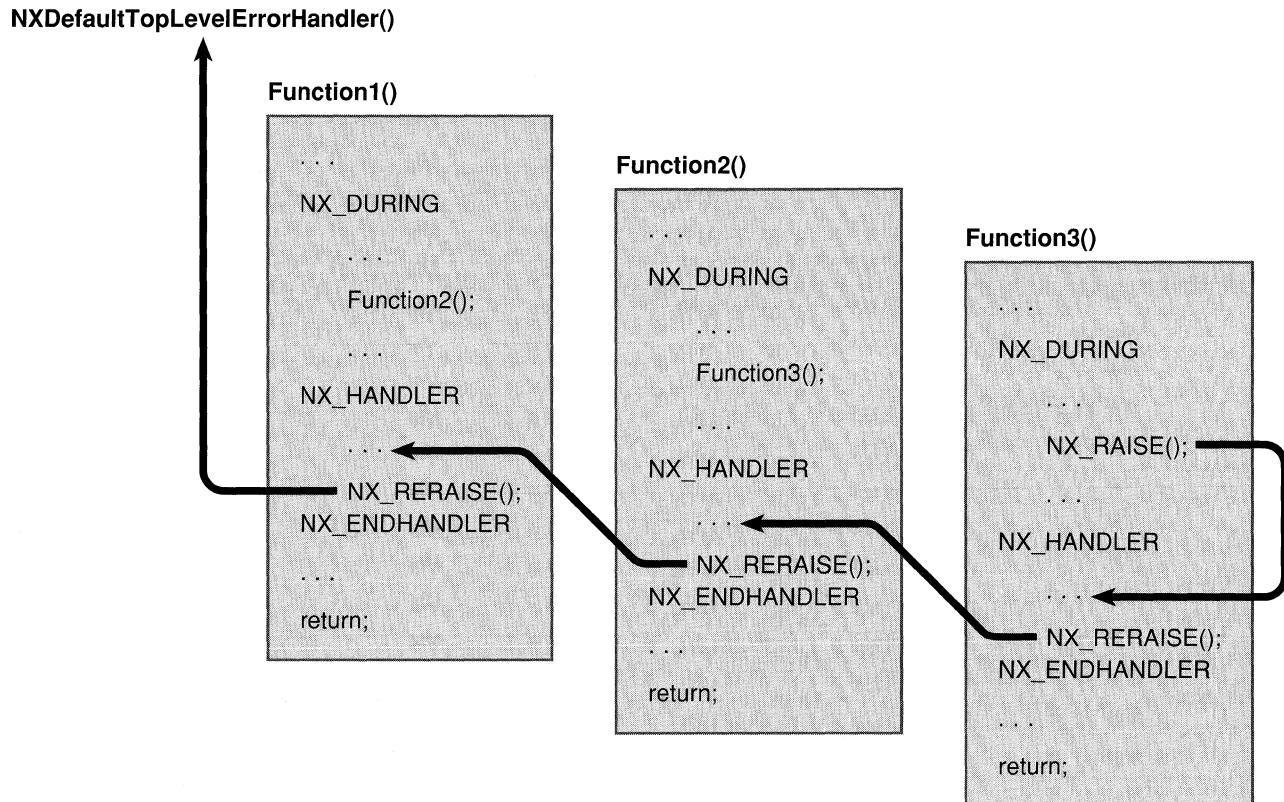


Figure 10-2. Nested Exception Handlers

An exception raised within **Function3()**'s domain causes execution to jump to its local exception handler. In a typical application, this exception handler checks the values contained in **NXLocalHandler** to determine the nature of the exception. For exception types that it recognizes, the local handler responds and then calls **NX_RERAISE()** to pass notification of the exception to the handler above it, in this case, the handler in **Function2()**. **Function2()**'s exception handler does the same and then reraises the exception to **Function1()**'s handler. Finally, **Function1()**'s handler reraises the exception. Since there's no exception handling domain above **Function1()**, the exception is transferred to the default top-level error handler, as discussed below.

An exception that's reraised appears to the next higher handler just as if **NX_RAISE()** had been called within its own exception handling domain. (**NX_RERAISE()** is in effect a cover for **NX_RAISE()**, transferring the values of **NXLocalHandler**'s **code**, **data1**, and **data2** members to the next higher exception handler.)

For applications based on the Application Kit, exceptions that are reraised within the highest-level local exception handler are sent to **NXDefaultTopLevelErrorHandler()**. Through a call to **NXReportError()**, this function prints a message about the exception. (See “Reporting Errors” below for more information.) If an application’s connection to the Window Server becomes corrupt or dies, or if the application is unable to form a connection to the Server, **NXDefaultTopLevelErrorHandler()** terminates the application by calling **exit()** with a status code of -1.

NXSetTopLevelErrorHandler() lets you change the function used as the top-level handler; **NXTopLevelErrorHandler()** returns a pointer to the current top-level handler. If you substitute your own function for **NXDefaultTopLevelErrorHandler()**, you should probably call **NXDefaultTopLevelErrorHandler()** as part of its implementation. In this way, your function can give special handling to certain exceptions, passing all others to **NXDefaultTopLevelErrorHandler()**.

Raising an Exception Outside of an Exception Handler

If an exception is raised outside of any exception handler, it’s intercepted by the *uncaught exception handler*, a function set by **NXSetUncaughtExceptionHandler()** and returned by **NXGetUncaughtExceptionHandler()**. The default uncaught exception handler for Application Kit programs writes the message “An uncaught exception was raised.” to the Workspace Manager’s console window (if the application was launched by the Workspace Manager) or to a Shell or Terminal window (if the application was launched from either of those applications). It then calls the top-level exception handler, passing it the information contained in the arguments to the **NX_RAISE()** call that originally raised the exception.

You can change the way uncaught exceptions are handled by using **NXSetUncaughtExceptionHandler()** to establish a different procedure as the handler. However, because of the design of the Application Kit, it’s rare for an exception to be raised outside of an exception handling domain. The Application object’s event loop itself is within an exception handling domain. On each cycle of the loop, the Application object retrieves an event and sends an event message to the appropriate object in the application. Thus, the code you write for custom objects (as well as the code for Application Kit objects) is executed within the context of the event loop’s exception handler. To customize the Application Kit’s highest-level response to exceptions, modify the top-level exception handler.

Exception Codes

Each of the software modules provided by NeXT is assigned a range of exception code values. The lowest value within the range is represented by a constant, as listed in the following table.

Exception Category	Base Constant
Client Library, Adobe	DPS_ERRORBASE
Client Library, NeXT Extensions	DPS_NEXTERROREBASE
Application Kit	NX_APPKITERRBASE
Streams	NX_STREAMERRBASE
Typed Streams	TYPEDSTREAM_ERROR_RBASE
Music Kit	MK_ERRORBASE
DSP C Library	DSP_ERRORS

Except for the first two categories, each range spans 1000 exception codes. The first two categories share a range of 1000: The first 100 codes are reserved for exceptions generated by Adobe's client library routines; the remaining 900 codes are reserved for the NeXT client library.

If, within an exception handler, you want to catch exceptions from one of these modules, you could use code such as this:

```

NX_HANDLER
{
    if (NXLocalHandler.code >= DSP_ERRORS &&
        NXLocalHandler.code < DSP_ERRORS +1000) {
        /* code for custom handling of DSP exceptions */
    } else
        NX_RERAISE();
}
NX_ENDHANDLER

```

Defining Codes for Your Application

The Application Kit defines one additional range of exception codes: the range for applications built on the Application Kit. This range extends upward from the base constant NX_APPBASE, as defined in the header file **appkit/errors.h**. For example, you could use this constant in an enumeration of exception types for a database application:

```

enum databaseExceptions {
    DB_invalidSearchKey = NX_APPBASE,
    DB_corruptRecord,
    DB_corruptIndex,
    DB_compactionError,
    DB_sortError
};

```

By initializing exception codes in this way, you can be sure that they won't conflict with those assigned to other software modules.

Associating Messages with Codes

In general, you'll want to associate a message with each exception code you define for an application. One convenient way to centralize this list of messages is to declare an array of pointers to the messages that correspond to the application-specific exception codes:

```
char *dbErrorMessages[] = {
    /* DB_invalidSearchKey */     "Invalid search key",
    /* DB_corruptRecord */       "Error reading record",
    /* DB_corruptIndex */        "Error opening index",
    /* DB_compactionError */     "Error during compaction",
    /* DB_sortError */           "Error during sort operation"
};
```

Using this technique, a call to **NX_RAISE()** might look like this:

```
NX_RAISE(DB_corruptIndex,
         dbErrorMessages[DB_corruptIndex -NX_APPBASE], NULL);
```

The first argument is the exception code, and the second is a pointer to the string “Error opening index”.

Besides centralizing your application’s error messages, associating exception codes and messages in this way makes it easier for your application to work with the Application Kit’s error reporter, as described in the next section.

Reporting Errors

The Application Kit lets you register a function as the error reporter for a range of exception codes. An error reporter typically logs information about the error by writing a message in the Workspace Manager’s Console window. An application can have many error reporters, each responsible for a specific range of exception codes.

Once the reporters are registered (as described below), calling **NXReportError()** causes the Application Kit to search for the reporter responsible for the specific exception code. If it finds one, the reporter is called and passed data about the exception. If it can’t find one, it logs the exception code and a message stating that an unknown exception code was reported.

An error reporter for the database application example introduced above might look like this:

```
void
DBErrorReporter(NXHandler *errorCode)
{
    if (errorCode->code == DB_invalidSearchKey)
        return;
    NXLogError("DB error: %s\n",
               dbErrorMessages[errorCode->code -NX_APPBASE]);
    return;
}
```

This reporter ignores exceptions of type **DB_invalidSearchKey** (presumably because they are recoverable errors) but logs messages concerning all other codes within its range.

The function **NXLogError()** is much like **printf()**: It lets you write a formatted string to the Console, Shell, or Terminal window, depending on where the application was launched. **NXLogError()**, however, calls **syslog()**, which marks the message with the time of occurrence and the application's process identification number. See the UNIX manual page for **syslog()** for more information.

If your application defines a range of exception codes as its own, it should also register an error reporter. This is because the default top-level exception handler calls **NXReportError()** for all exceptions raised to its level. If your application hasn't registered an error reporter for its range, then **NXReportError()** won't be able to print anything more informative than the exception code for the error.

Registering Error Reporters

You register an error reporter for a range of exception codes by calling **NXRegisterErrorReporter()**. You might, for example, place code such as the following in the **initialize** method of one of your application's custom classes:

```
+ initialize
{
    NXRegisterErrorReporter(NX_APPBASE, NX_APPBASE + 999,
                           DBErrorReporter);
    return self;
}
```

The first two arguments to **NXRegisterErrorReporter()** represent the minimum and maximum values for exception codes sent to the reporter referred to by the third argument. The call above specifies the error reporter described in the previous section. Once an error reporter is registered for a specific range, no new reporter can be set for any part of that range until the existing reporter is removed.

You remove an error reporter by calling **NXRemoveErrorReporter()**. This function takes one argument, the minimum exception code value of the existing error reporter's range. For example, to remove the reporter registered in the preceding code excerpt, you'd make this call:

```
NXRemoveErrorReporter(NX_APPBASE);
```

Handling PostScript Errors

To draw on the screen, your application must send PostScript code to the Window Server, where it's executed by the PostScript interpreter. Most of the code that an application sends is generated by user-interface objects defined in the Application Kit. Other code is generated by routines you write for your application's custom classes. In some applications (for example, a PostScript language previewer such as Yap—see [/NextDeveloper/Examples/Yap](#)), the code is entered by the user or imported from a file. Careful debugging should ensure that PostScript code generated by Application Kit or custom objects won't generate exceptions at run time. However, applications that import PostScript code must be prepared to handle exceptions raised by the PostScript interpreter at run time.

In the following example, a PostScript program is executed from a file. Before reading the file, however, the application stores an image of its own PostScript virtual memory (VM) in the PostScript dictionary **userdict**. It then begins importing and executing the code. If the code is faulty, the Display PostScript client library raises an exception, transferring control to the local exception handler. After the handler has logged the exception, the contents of the application's PostScript VM are restored from **userdict**.

```
char      *errorBuffer;
int       streamLength, maxLength;
NXStream *errorStream;

/* save contents of PostScript VM */
PSuserdict();
DPSPrintf(DPSGetCurrentContext(), "/saveToken");
PSSave();
PSput();

NX_DURING
PSrun(/* file path */);
NXPing(); /* ensure all code is transmitted to Server */
```

```

NX_HANDLER
    switch (NXLocalHandler.code) {
        case dps_err_ps:
            errorStream = NXOpenMemory(NULL, 0, NX_WRITEONLY);
            DPSPrintErrorToStream(errorStream, (DPSBinObjSeq)
                (NXLocalHandler.data2));
            NXFlush(errorStream);
            NXGetMemoryBuffer(errorStream, &errorBuffer,
                &streamLength, &maxLength);
            NXLogError("%s\n", errorBuffer);
            NXCloseMemory (errorStream, NX_FREEBUFFER);
            /* restore contents of PostScript VM */
            break;
        . . .
        default:
            NX_RERAISE();
    }
NX_ENDHANDLER

/* restore contents of PostScript VM */
PSuserdict();
DPSPrintf(DPSGetCurrentContext(), "/saveToken");
PSget();
PSrestore();

```

The exception code **dps_err_ps** identifies errors reported by the PostScript language interpreter in the Window Server. (See **dpsclient/dpsclient.h** for a complete list of exception codes raised by the Display PostScript client library.) The local exception handler treats exceptions of this type by calling **DPSPrintErrorToStream()** to extract the error message from the binary object sequence returned from the Window Server. It then prints the message to a stream. **NXGetMemoryBuffer()** gives a pointer to this message, which **NXLogError()** uses to log the error message, as described earlier. By default, all other exception types are reraised to upper-level handlers. If the default top-level exception handler receives a PostScript exception, it logs the corresponding error message in much the same way as illustrated here.

This excerpt contains a number of additional points of interest. First, note that **NXPing()** is called after the PostScript code is sent to the Window Server. Since an application and the Window Server are separate processes that execute asynchronously, notification of an error might not be received by the application until after control has passed from the exception handling domain. Calling **NXPing()** keeps the two processes synchronized, ensuring that exceptions generated by the PostScript code are raised within the exception handling domain.

Second, using **PSrun()** assumes that the file containing the PostScript program has the same path on all machines that run this application—a perilous assumption! **PSrun()** was used for simplicity in this example; a better choice for sending PostScript code to the Server is **DPSWriteData()**. (See the *Client Library Reference Manual* by Adobe Systems, Inc. for more information on **DPSWriteData()**.)

Finally, the method of protecting an application's VM illustrated here isn't foolproof. In particular, **restore** doesn't affect the contents of the dictionary stack. If an imported PostScript program removes dictionaries that your application placed on the stack, they won't be restored. Perhaps the best way to protect your application from errors caused by an imported PostScript code is to execute the code in a context separate from that of your application. Then, if the code generates errors, your application can respond by calling **DPSDestroyContext()**.

Managing Exception Data

The macro **NX_RAISE()** takes three arguments: an exception code and two pointers to data that you might pass to the exception handler. So far in this discussion, we've used one of these pointers to pass an error message that's associated with the exception (see "Associating Messages with Codes" above). You might, in some circumstances, need to pass additional data along to the exception handler. The functions **NXAllocErrorHandler()** and **NXResetErrorHandler()** help you manage the memory that you might allocate for this additional data.

NXAllocErrorHandler() controls a buffer for error data, allocating the amount of memory you request; **NXResetErrorHandler()** frees this memory. The Application Kit calls **NXResetErrorHandler()** each time through the event loop, making it unnecessary for you to call the function directly. (However, if your application doesn't use the Application object's event loop, be sure to call **NXResetErrorHandler()** after getting each event in order to free this memory.) Thus, by using **NXAllocErrorHandler()** whenever you need to allocate memory within your exception handler, you don't have to be concerned about freeing this memory when it's no longer needed.

NXAllocErrorHandler() takes two arguments: One specifies the amount of memory to allocate, and the other refers to a pointer to this memory. This code excerpt illustrates its use:

```
CheckSearchKey()
{
    char *errorData;
    char theKey[1024];

    NX_DURING
        /* get search string and initialize theKey */
        if /* invalid search key */ {
            NXAllocErrorHandler(strlen(theKey)+1, &(void *)errorData);
            strcpy(errorData, theKey);
            NX_RAISE(DB_invalidSearchKey,
                     dbErrorMessages[DB_invalidSearchKey - NX_APPBASE],
                     errorData);
        }
    . . .
}
```

```

NX_HANDLER
    switch(NXLocalHandler.code) {
        case DB_invalidSearchKey:
            NXRunAlertPanel (NXLocalHandler.data1,
                            NXLocalHandler.data2, "Restart",
                            NULL, NULL);
            NX_RERAISE();
        . . .
        default:
            NX_RERAISE();
    }
NX_ENDHANDLER
return;
}

```

In this example, if an exception occurs **NXAllocErrorHandler()** allocates memory for the search key that caused the exception. The key is then copied into the newly allocated memory. **NX_RAISE()** passes this data to the local exception handler where it's used as part of the text of an attention panel displayed to the user. From there, the exception is raised to the next higher-level handler.

We might have avoided allocating the storage represented by **errorData** by simply using **theKey** as the third argument to **NX_RAISE()**. However, **theKey** is only defined within the scope of this function; reraising the exception within the local exception handler exits this block, and so **theKey** would be undefined for higher-level exception handlers. Copying the search key into memory allocated with **NXAllocErrorHandler()** not only makes it available to the higher-level handler, but ensures that the memory will be freed when it's no longer needed.

Index

abstract superclass 3-10
action message 6-27, 6-37, 7-40
 in the responder chain 7-43
action method 7-42, 7-45
ActionCell class 6-28
active application 2-52, 7-17, 7-24
 changing 7-25
Align menu in Interface Builder 8-67
Alignment command in Interface Builder 8-67
alpha value 4-30
ancestor of a View 6-16
application
 dock 2-30
 framework 6-37
 name 6-64
 setup 7-9
 structuring 6-8
application-activated event 5-8
application-deactivated event 5-8
application-defined event 5-8, 6-37, 7-35
 event data 5-17
Application Kit 1-6, 6-3
 conventions 6-68
 inheritance hierarchy 6-4
 principal classes 6-12
 printing architecture 7-107
Application object 6-33
 services 6-34
archiving 10-11
 arbitrary data 10-12
 arrays 10-13
 a data structure 10-11
 id instance variables 10-15
 and the Object class 3-27
 objects 10-14
 a Text object 9-34
 to a typed stream 10-11
ascender 9-13
attached submenu 2-61
attention panels 2-76
Attributes display 8-81

background
 color 4-28, 6-54
 View 7-93

base coordinate system 4-10
baseline 9-13
bitmap 4-33
body rectangle 9-5
border
 of a Box 9-37
 of a window 2-39, 4-12
bounds instance variable 6-23
bounds rectangle 6-23
Box object 9-35
 attributes 8-90
 contents 9-40
 creating 9-36
 modifying 9-36
 resizing 9-41
break table 9-14, 9-28
Bring to Front command in Interface Builder 8-65
buffer
 application-Window Server 4-50
 window 4-23
buffered window 4-23
button 2-83
 attributes 8-83

calculator application example 8-25
 modifying 8-35
caret 9-15
Cell class 6-27
character
 code 5-15, 5-17
 filter function 9-6, 9-29
 keys 2-12
 layout 9-13
 position 9-5
 set 5-15, 5-17
 wrapping 9-14
class 3-7
 defining 3-10
 implementation 3-13
 interface 3-11
 method 3-11
 object 3-7, 3-16
Class Inspector 8-101
Classes icon in File window 8-73
Clear compositing operation 4-41

click 2-17
 in a window 2-59
 click table 9-28
 clipping 7-81
 conventions 4-60
 close button 6-54
 Close File command in Interface Builder 8-58
 Close Window command in Interface Builder 8-64
 color
 background 4-28, 6-54
 on the NeXT screen 4-7
 setting 4-30
 value 4-33
 Command key-down event 7-50
 command-line arguments 6-64
 commands 2-64
 standard 2-67
 comments in Objective-C 3-36
composite operator 4-36
compositerect operator 4-38
 compositing 4-29, 4-36
 PostScript operators 4-36
 type of operations 4-39
 compound event 5-5
 content
 area 2-39, 4-12
 rectangle 6-29, 6-62
 view 6-30
contentView instance variable 6-30, 6-41
 Control class 6-26
 control panels 2-79
 controls 2-9, 2-81
 coordinates 4-6
 converting 7-79
 cursor 7-36
 flipping 7-71
 modifying 4-11, 7-75
 printing 4-13
 screen 4-9
 in the superview 7-82
 in a View 7-70
 View 4-11, 6-21
 window 4-10, 6-31
 Copy command in Interface Builder 8-61
 Copy compositing operation 4-41
 coverage 4-4
 setting 4-30
 CTM 4-21
 current coordinate system 4-11
 current window 4-21
 cursor 2-22
 changing 7-53
 coordinates 7-36
 defining 7-53
 hiding 2-23
 rectangle 7-54
 wait cursor 7-56
 cursor-update event 5-7, 7-16
 CustomView attributes 8-93
 Cut command in Interface Builder 8-61
 defaults 10-21
 global 10-27
 reading 10-23
 system 10-26
 Workspace Manager 10-30
 writing 10-23
 deferred window 6-54
@defs() directive 3-34
 delegate 6-8, 6-45
 Delete command in Interface Builder 8-62
 descendant of a View 6-16
 descender 9-13
 descent line 9-13
 dictionary stack 4-21
 discrete event 5-5
 dispatch table 3-17
 display methods 7-90, 7-95
 arguments 7-92
 Display PostScript 1-7, 4-3
 contexts 7-110
dissolve operator 4-43
 double-click 2-18
 drag 2-19
 drawing 4-3
 coordinates 6-21
 instance drawing 4-45
 text 7-74
 and the Text class 9-25
 a View 7-86
 in the view hierarchy 7-69
 dynamic
 allocation 3-29
 binding 3-6, 3-17
 typing 3-29
 Edit menu 2-71
 in Interface Builder 8-60
 editor application example 8-44
@encode() directive 3-34
 encoding vector 5-17
 error reporter 10-45

event 1-7, 5-3
 data 5-13
 flags 5-11
 handling 7-5
 loop 7-5, 7-57
 mask 5-18, 7-14
 message 7-27, 7-39
 queue 5-20
 reading 7-60
 record 5-9
 type constants 5-10
 types 5-4
exception
 codes 10-43
 data 10-49
 detection 10-37
 handling 10-36, 10-39
 raising 10-39

factory object 3-7
file
 descriptor 5-24
 names 2-35
 opening 7-12
 package 2-37
 system 2-31

File menu in Interface Builder 8-55

File window in Interface Builder 8-69

File's Owner
 attributes 8-96
 icon 8-70

filter function 9-6, 9-26, 9-28

Find menu 2-74

first responder 6-26, 6-31, 7-18
 changing 7-19

First Responder icon 8-71

flags-changed event 5-5

flushWindow method 7-100

focusing on a View 7-80

font 9-24

Font menu 2-73
 in Interface Builder 8-62

FontManager object in File window 8-71

Form attributes 8-88

FormCell attributes 8-89

frame instance variable
 of a View 6-17
 of a Window 6-29

frame rectangle
 of a Text object 9-6, 9-11
 of a View 6-17, 7-102
 of a Window 6-29, 6-62

frame view 6-30

graphics state 4-21
 changing 4-22
 object 4-22
 parameters 7-88
 stack 4-21

gray value 4-30

Group command in Interface Builder 8-65

half-open shapes 4-59

halftone 4-6, 4-8

Hide File command in Interface Builder 8-58

Hide Grid command 8-68

highlighting in images 4-44

home directory 2-32, 6-64

host name 6-63, 6-66

icon 2-43
 attributes 8-96
 header file 8-9

Icons in File window 8-72

id data type 3-4

imaging conventions 4-51

@implementation directive 3-13

#import directive 3-12

inheritance 3-8

inheritance hierarchy 3-28
 Application Kit 6-4

Inspector command in Interface Builder 8-63

Inspector window in Interface Builder 8-80
 Attributes display 8-81
 Autosizing display 8-99
 Class display 8-101
 Connections display 8-98
 Miscellaneous display 8-100
 Project display 8-101

instance 3-7
 drawing 4-45
 method 3-11

instance variable 3-3
 inheriting 3-9
 public 3-31
 reading and writing 6-68

interface 2-5
 file 8-7
 testing 8-21, 8-60

Interface Builder 1-5, 8-5
 project 8-8
 reference 8-53
 tutorial 8-11

@interface directive 3-11

intersection of rectangles 4-15

isa instance variable 3-17

key code 5-15, 5-18
 key-down event 5-5
 event data 5-15
 key equivalent 7-49
 key-up event 5-5
 event data 5-15
 key window 2-55, 7-17, 7-21
 changing 7-22
 choosing 2-58
 keyboard 2-11
 alternatives 2-24, 7-49
 event 5-5
 events 7-16, 7-29
 information 5-17
 kit-defined event 5-7, 7-34
 event data 5-16

Layout menu in Interface Builder 8-64
 line layout 9-13
 Listener class 6-34

Mach 1-8
 port 5-24, 10-10
 main event loop 7-5
 main menu 2-66
 main window 2-56, 7-17, 7-21
 choosing 2-58

Make Column command 8-68
 Make command in Interface Builder 8-59
 Make Row command 8-67
 makefile 8-9, 8-22
 Matrix attributes 8-85
 MegaPixel Display 4-7
 memory stream 10-8
 menu 2-60
 updating 7-97
 Menu palette 8-78
 MenuCell attributes 8-95
 message 3-4
 expression 3-4, 3-36
 messaging 3-16
 errors 3-26
 method 3-3
 adding 3-15
 address 3-33
 defining 3-14
 inheriting 3-9
 overriding 3-10, 6-10, 6-12
 selector 3-4, 3-19

MIDI 1-10
 miniwindow 2-43, 6-52

modal
 event loop 7-57
 sessions 7-67
 windows 7-66
 modifier keys 2-14
 and the mouse 2-21

mouse 2-17
 buttons 2-23
 and modifier keys 2-21
 priority 2-26
 scaling 2-22
 mouse-down event 5-5
 event data 5-14
 mouse-dragged event 5-5
 mouse-entered event 5-6, 7-32
 event data 5-15
 mouse events 5-5
 coordinating 7-58
 left 7-29
 right 7-32
 mouse-exited event 5-6, 7-32
 event data 5-15
 mouse-moved event 5-6
 mouse-up event 5-5
 event data 5-14
 Music Kit 1-10
 music overview 1-9

named object 6-49
Net directory 2-34
 New Application command 8-56
 New Module command 8-56
 next responder 6-15
nextResponder instance variable 6-43
 NeXTstep 1-5
 nonretained window 4-23
 NXApp global variable 6-33
 NXStream structure 10-20

object 3-3
 adding 8-18
 data structure 3-33
 editing 8-16
 Object class 3-25
 archiving support 3-27
 initialization 3-26
 memory management 3-26
 object-oriented programming 3-3
 Objective-C 3-3
 language synopsis 3-36
 off-screen window 4-27
 one-shot window 6-54
 opaque View 7-94
 Open command in Interface Builder 8-55

operand stack 4-21
 operating system *See* Mach
 outlet 6-44
 defining 6-49

Page Layout command in Interface Builder 8-64
 pagination 7-109
 Palettes command in Interface Builder 8-63
 Palettes window in Interface Builder 8-76
 panels 2-75
 Paste command in Interface Builder 8-61
 pasteboard 10-31
 data types 10-31
path environmental variable 2-34
 pixel 4-6
 value 4-33
 PlusD compositing operation 4-42
 PlusL compositing operation 4-42
 polymorphism 3-6
 pop-up list 2-44
 posing in Objective-C 3-29
 PostScript 4-3
 code generating 7-106
 error handling 10-47
 execution context 4-20
 functions 4-49
 sending code to Window Server 4-47
 power-off event 5-8
 press 2-21
 printing 7-106
 coordinates 4-13
 panels 7-110
 Project command in Interface Builder 8-58
 Project Inspector 8-102
 Attributes display 8-103
 Files display 8-106
 project in Interface Builder 8-8
 creating 8-14
pswrap program 4-48
@public directive 3-32
 public instance variables 3-31
 pull-down list 2-45

read: method 10-17
 receiver of a message 3-4
 rectangles 4-14
 registration table 10-21
 remote message 6-9, 6-37
 Request menu 2-75
 Resize Window command in Interface Builder 8-66
 resolution 4-7
 responder chain 6-15
 and action messages 7-43
 and event messages 7-39

Responder class 6-14
 retained window 4-23
 Rich Text Format 9-35
 RTF 9-35

Same Size command in Interface Builder 8-65
 sample of a sound 1-9
 Save All command in Interface Builder 8-58
 Save As command in Interface Builder 8-58
 Save command in Interface Builder 8-57
 scheduling procedures 5-25, 7-61
 scorefile 1-10
 ScoreFile language 1-10
 screen 4-6
 coordinate system 4-9
 list 4-26, 6-62
 size 6-63
 scroll buttons 2-90
 scroller 2-88
 ScrollView attributes 8-91
 segments 6-66
 Select All command in Interface Builder 8-62
 selection 2-26
 continuous extension 2-28
 discontinuous extension 2-29
 selector 3-4, 3-19
@selector() directive 3-19
self 3-21
 redefining 3-24
 returning 6-71
 Send to Back command in Interface Builder 8-65
 Set Grid Off command 8-68
 Set Grid On command 8-68
 Show Grid command 8-68
 single-operator functions 4-49
 Size to Fit command in Interface Builder 8-65
 slider 2-82
 attributes 8-91
 smart cut and paste 9-22
 tables 9-27
 sound
 adding to an application 8-42
 attributes 8-97
 overview 1-9
 Sound Kit 1-9
 soundfile 1-9
 Sounds icon in File window 8-73
 Speaker class 6-34
 standard windows 2-38
 static typing 3-30
 stream 10-4
 and the pasteboard 10-33
 subclass 3-8, 6-7
 defining 6-70

submenu 2-61
detaching 2-63
hierarchy 2-64
subview 6-16
resizing 7-103
subviews instance variable 6-41
super 3-21, 3-23
superclass 3-8
abstract 3-10
supermenu 2-61
superview 6-16
superview instance variable 6-41
system control keys 2-14
system-defined event 5-8, 7-36
 event data 5-17
system overview 1-3

tag 6-71, 7-47
 Text object 9-21
target 6-27, 6-48, 7-41
Test Interface command 8-60
text
 alignment 9-12
 editability 9-18
 editing 9-18
 examining 9-9
 field 2-86
 filter function 9-6, 9-31
 font 9-24
 layout 9-11
 margins 9-12
 selection 2-29, 9-15
 setting 9-8
 tables 9-26
 writing to a file 9-10
text field 2-86
 attributes 8-81
Text object 9-4
 archiving 9-34
 creating 9-6
 and drawing 9-25
 reusing 9-34
 in a ScrollView 9-32
text:isEmpty: 9-19
textDidChange: method 9-19
textDidEnd:endChar: 9-20
textDidResize:oldBounds:invalid: 9-21
textWillChange: method 9-19
textWillEnd: method 9-20
textWillResize: method 9-21
tiers of windows 4-26
timed entry 5-22
timer event 5-6, 7-16, 7-63

title
 bar 2-39, 6-53
 of a Box 9-38
tracking rectangle 5-6, 5-15
transformation matrix 4-21
transparency 4-29
 compositing operations 4-42
triple-click 2-18
type encoding 3-34
typed stream 10-11
 closing 10-19
 on a file 10-19
 on memory 10-20
 opening 10-19
 and the pasteboard 10-34
types in Objective-C 3-36

unarchiving
 a data structure 10-16
 objects 10-17
uncaught exception handler 10-43
Ungroup command in Interface Builder 8-66
union of rectangles 4-15
user interface 2-5
 objects 9-3
user name 6-64

View class 6-15
 subclasses 6-26
view hierarchy 6-16
 of a Box 9-40
 drawing in 7-69
 managing 6-30
View object
 clipping 7-81
 coordinate 4-11
 coordinate systems 7-70
 displaying 6-24, 7-90
 focusing 7-80
 updating 7-98
View palette 8-77
visible rectangle 6-24

wait cursor 7-56

window 2-37
 attributes 8-93
 behavior 2-47
 buffering 4-23
 closing 2-50
 coordinates 4-10
 displaying 7-96
 hiding 2-51
 information 6-61
 list 6-33
 managing 6-50
 miniaturizing 2-50
 moving 2-49, 6-57
 number 4-20, 6-62
 off-screen 4-27
 ordering 2-46
 placement 2-47
 reordering 2-48, 6-60
 size 2-45, 2-49, 6-59
 status 6-56
 style 2-40
 tiers 4-26
 types 2-38
Window class 6-29
 subclasses 6-32
window-exposed event 5-8, 7-34
window instance variable 6-42
Window menu 2-69
window-moved event 5-7, 7-34
Window palette 8-78
window-resized event 5-7, 7-35
Window Server 1-6, 4-47
window system 4-19
windowList instance variable 6-33, 6-40
Windows menu in Interface Builder 8-63
word wrapping 9-14
Workspace Manager defaults 10-30
workspace window 2-39

x-height of a font 9-13

zero-width lines 4-57

NeXT Computer, Inc.
900 Chesapeake Drive
Redwood City, CA 94063

Printed in U.S.A.

2908.00

12/90

Text printed on
recycled paper

