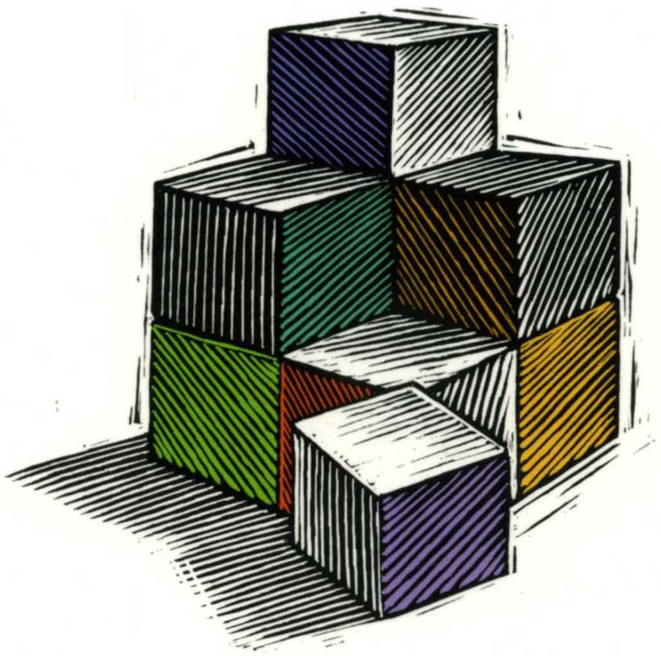
DEVELOPMENT TOOLS AND TECHNIQUES



NEXTSTEP™

Object-Oriented Software

NeXTSTEP™ DEVELOPMENT TOOLS AND TECHNIQUES

NeXTSTEP Developer's Library
NeXT Computer, Inc.

Release 3

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York • Don Mills, Ontario
Wokingham, England • Amsterdam • Bonn • Sydney • Singapore • Tokyo • Madrid
San Juan • Paris • Seoul • Milan • Mexico City • Taipei

This manual describes NeXTSTEP Release 3.

Written by NeXT Publications.

This manual was designed, written, and produced on NeXT computers. Proofs were printed on a NeXT 400 dpi Laser Printer and NeXT Color Printer. Final pages were transferred directly from a NeXT optical disk to film using NeXT computers and an electronic imagesetter.

3 4 5 6 7 8 9 10–CRS–96959493
Third printing, November 1993

ISBN 0-201-63249-7

Contents

Introduction

NeXTSTEP™ Development Tools and Techniques describes the essential tools for developing a NeXTSTEP application—these tools include the Project Builder, Interface Builder™, Terminal, and Edit applications, miscellaneous developer applications, and the GNU C compiler, preprocessor, and debugger. The manual is part of a collection of manuals called the *NeXTSTEP Developer's Library.*

This manual assumes you're familiar with the standard NeXTSTEP user interface. Experience using a variety of NeXTSTEP applications would also be helpful. Some topics that are discussed here aren't covered in detail; instead, you're referred to a generally available book on the subject, or to an on-line source of the information.

A version of this manual is stored on-line in the NeXT™ Digital Library (which is described in the *User's Guide*). The Digital Library also contains release notes, which provide last-minute information about the latest release of the software.

How This Manual is Organized

The first 14 chapters of this manual concentrate on the tools used in building a NeXTSTEP application. The last four chapters contain step-by-step instructions for creating several simple applications, thereby providing a hands-on overview of the application development process.

- Chapter 1, "Putting Together a NeXT Application," provides an overview of the tools and techniques that you'll use to assemble a working application. The tools introduced in this chapter are discussed in greater detail in other chapters of this manual.
- Chapter 2, "The Project Builder Application," describes the central control point for application development in NeXTSTEP. Project Builder helps you with each stage of application development, from inception to installation.
- Chapter 3, "The Interface Builder Application," describes the tool that lets you assemble your application's user interface (and other parts) from predefined building blocks, and lets you create new building blocks of your own design.
- Chapter 4, "The Edit Application," describes the NeXTSTEP text editor you'll be using to edit and debug your application's source files.
- Chapter 5, "The Terminal Application," describes the application you'll use to interact with a UNIX® shell from the NeXTSTEP workspace.
- Chapter 6, "The Icon Builder Application," describes a simple graphic editor for creating and editing application icons.
- Chapter 7, "The DBModeler Application," describes an application for building data models based on the structure of an existing relational database. The resulting models can be used in Interface Builder to construct applications that access the data in the database.
- Chapter 8, "The MallocDebug Application," describes an application for measuring the dynamic memory usage of the applications you develop.
- Chapter 9, "The Process Monitor Application," describes an application that lets you examine running processes, and pause or kill any of the processes and applications running on your computer.

- Chapter 10, "The PostScript® Previewers Yap and pft," describes two tools: an application for developers who want to write and test PostScript code, and a shell-based interface to the PostScript Window Server.

- Chapter 11, "The GNU C Compiler," describes GNU CC, the ANSI-standard C compiler used on NeXT computers. The chapter also describes how to compile a C program using the GNU compiler.

- Chapter 12, "The GNU C Preprocessor," describes the macro preprocessor that's used to transform your C program or application before actual compilation. The chapter provides information about standard and precompiled header files, macros, and conditionals. It also lists the options that can be used with the **cpp** (C preprocessor) command.

- Chapter 13, "The GNU Source-Level Debugger," describes GDB, the primary tool you'll use to debug the applications that you develop.

- Chapter 14, "Mach Object Files," describes the format of Mach object (also known as Mach-O) files, which NeXT computers use instead of the UNIX 4.3BSD **a.out** format.

- Chapter 15, "Building a Simple Application," provides a tutorial introduction to the process of application development in NeXTSTEP. It gives you an introduction to Project Builder and Interface Builder, while showing you some of the basic features of the Application Kit.

- Chapter 16, "Building a One-Button Calculator," continues the tutorial introduction to the major development tools in NeXTSTEP and gives you further insight into object-oriented programming with the Application Kit™.

- Chapter 17, "Building a Text Editor Using Multiple Nib Files," shows how Interface Builder and the Application Kit are used to tackle more advanced issues of object-oriented application design.

- Chapter 18, "Building a Custom Palette," the final tutorial in the series, shows you how Interface Builder itself can be modified to include the objects and tools you find most useful.

Conventions

Syntax Notation

Where this manual shows the syntax of a function, command, or other programming element, the use of bold, italic, square brackets, and ellipsis has special significance, as described here.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

print *expression*

means that you follow the word **print** with an expression.

Square brackets [ ] mean that the enclosed syntax is optional, except when they're bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

pointer [*filename*]

means that you type a pointer with or without a file name after it, but

[*receiver message***]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous syntax element may be repeated. For example:

Syntax	Allows
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

Special Characters

In general, notation like

Alternate-x

represents the character you get when you hold down the Alternate key while typing x. Because the modifier keys Alternate, Command, and Control interpret the case of letters differently, their notation is somewhat different:

Notation	Meaning
Alternate-x	Hold down Alternate while typing lowercase x.
Alternate-X	Hold down Alternate while typing uppercase X (with either Shift or Alpha Lock).
Alternate-Shift-x	Same as Alternate-X.
Command-d	Hold down Command while typing lowercase d; if Alpha Lock is on, pressing the D key will still produce lowercase d when Command is held down.
Command-Shift-D	Hold down Command and Shift while pressing the D key. Alpha Lock won't work for producing uppercase D in this case.
Control-X	Hold down Control while pressing the X key, with or without Shift or Alpha Lock (case doesn't matter with Control).

Notes and Warnings

Note: Paragraphs like this contain incidental information that may be of interest to curious readers but can safely be skipped.

Warning: Paragraphs like this are extremely important to read.

1

Putting Together a NeXTSTEP Application

1

Putting Together a NeXTSTEP Application

There are a number of ways you might draw the line between programs and applications. Programs are simple; applications are complicated. Programs are small; applications are big. Programs run from a command line; applications have a graphic user interface. A program has just a few source files; an application may have lots and lots.

No matter how you draw the line, as you move from writing programs to developing applications, you need to focus increasing attention on project management. If the application is the end result, the project is how you get there. The project can be thought of as both the steps you go through and the source files you use to construct an application.

A complete project management strategy includes strategies for creating, organizing, and maintaining source files, building the application from its sources, running and debugging the application, revising the source files to fix bugs, and installing the finished application—or preparing it for others to install.

In NeXTSTEP, the hub of application development is Project Builder—a project manager that is itself a NeXTSTEP application. Project Builder isn't the only tool you use to manage your project and develop your application. Instead, it's like a control center from which you switch from one application development task to another, and from one tool to another.

This chapter takes a brief look at the components of a NeXTSTEP application. It explains the path that Project Builder and other NeXTSTEP tools offer you for going from a set of source files to a working application. It looks at the application development process in terms of resources and tasks that you, the developer, must provide and those that Project Builder and other NeXTSTEP tools provide for you. Subsequent chapters present detailed reference for each of the tools introduced here. The last four chapters present step-by-step tutorials that offer you a chance to explore the NeXTSTEP development process for yourself.

The Application Development Process

The process of developing an application can be divided into three general tasks: designing, coding, and debugging. These tasks are never performed entirely sequentially. You may decide after some coding that you need to change some aspect of design. Debugging always reveals code that needs rewriting, and occasionally exposes design flaws. When you develop an application with NeXTSTEP, you can move easily among these tasks.

The following sections enumerate the components of the NeXTSTEP application development process, describing those portions for which you're responsible and those which Project Builder, Interface Builder, and other NeXTSTEP development tools handle for you. For more information on Project Builder, see Chapter 2; for more on Interface Builder, see Chapter 3.

Design Your Application

Before you write any code, you should spend some time thinking about design. Some components of application design to consider are functionality, program structure, and user interface. You should think about the goals of your application and the techniques you might use to meet those goals. You should determine the unique classes that your application will require and think about how to divide your program into separate modules. You should sketch out user interface ideas, and use Interface Builder to prototype and test those ideas.

Create a Project

With the basic design determined, you can use Project Builder to start a new project.

In NeXTSTEP, a project is physically represented by a directory under the control of Project Builder; all of the components of the project must reside in this directory. When you start a new project, Project Builder automatically generates the project directory and a set of source files common to all applications, including a main file, a nib file, a makefile, and others. The main file includes the standard **main()** function required in all C programs. The nib file is used by Interface Builder to archive the application's user interface (nib is an acronym for "NeXTSTEP Interface Builder"). The makefile is updated by Project Builder to keep track of all the source files from which your application is built. Another file in the project directory, **PB.project**, is used by Project Builder itself to keep track of various project components.

Throughout the life of the project, you will add to and update the files in the project directory. NeXTSTEP development tools, including Project Builder and Interface Builder may add to and maintain other files in this directory as your project grows.

Write Code for Your Application

To establish the unique workings of your application, you create class interface and implementation files that include code for the appropriate methods and instance variables. Interface Builder can help in this process by creating skeletal code for a class if you list the methods in the Inspector panel. If you create the source files first, Interface Builder can parse them to learn about their **id** instance variables and action methods.

Project Builder lets you add source files to your project at any time. You can create other source files using standard C, Objective C, and C++ code. Project Builder can also know about and manage other files, such as **pswrap** files containing PostScript code within C function wrappers.

Connect Objects with Interface Builder

In Interface Builder, you can interconnect objects in your application. For example, you can establish the target and action for a control in the interface.

Interface Builder puts information about the classes used by your application in the nib file; included are Application Kit classes and other classes provided by NeXTSTEP, as well as the custom classes you define. The nib file contains all the information required to generate the objects in your application at run time: specifications for objects, connections between objects, icons, sounds, and other features. A NeXTSTEP application can have one or more nib files for each application you create.

Add Other Resource Files

Resource files are frequently used to customize the user interface for your application. Project Builder allows you to add icons for both your application and its documents. Interface Builder allows you to add icons and sounds for the buttons in your user interface. You can put other images in your application using Application Kit classes and PostScript code. You can add other sounds using Sound Kit™ methods. Project Builder provides a drag-and-drop interface for adding sounds, images, and other resource files to your project, including unique icons for your application and its document files.

Choose Document Extensions for Your Application

If your application reads and writes documents, you'll need to take measures to see to it that the Workspace Manager™ knows about and can work with those files. First, you need to write file management code that saves the documents with a unique extension. You also need to use the Project Builder application's Attributes display to specify document extensions for an application. Project Builder adds these extensions to the appropriate file to assure that your application is invoked by Workspace Manager when the user double-clicks a file with the specified extensions.

If you plan to distribute your software, or want to avoid future collisions with file extensions used by other applications, register the document file extensions with the NeXT Extension Registry. A list of currently registered names and the address for the extension registry is included in the *User Interface Guidelines.*

Compile Your Program

As you add source files to your application, Project Builder lists them in the project makefile. When you use its Build command, Project Builder starts the **make** program which in turn reads the project makefile and generates the executable file from the sources. As **make** runs, it issues system commands to compile and link your application's source files into an executable file. The project Makefile, generated by Project Builder, provides the information **make** needs to do this job. The warnings generated by the compiler and link editor provide information to help you locate and fix bugs detected at compile time.

In building your project, **make** keeps track of source updates. Each time you run **make,** only the source files that have been updated since the last **make** are regenerated; the rest are used as is. This minimizes the time required to generate your executable file.

Once you start building your application, Project Builder provides an interactive interface to Edit for locating source code problems detected by the compiler and link editor. Anytime the compiler encounters an error, Edit can locate the code with a single click—you can then edit out the problem and begin compiling again.

Debug Your Program

After you successfully compile your program, you're ready to try running it. The easiest way to do so is by choosing Debug in the Project Builder application's Builder display. This selection builds your application (if necessary), then starts GDB in a Terminal shell. You can then run your application with GDB in a couple of ways:

- Use Edit's Gdb panel to step through your application while looking at the code being executed. The Gdb panel provides an easy-to-use, interactive interface that integrates GDB and Edit; it's described in Chapter 2, "The Project Builder Application."
- Run the program from the Terminal shell by issuing GDB commands. The GDB debugger and its commands are described in Chapter 13, "The GNU Source-Level Debugger."

Along with the compiler and GDB, the NeXTSTEP development environment includes several applications and features that can help you trace your program and pinpoint errors. Other developer applications—including MallocDebug (Chapter 8), ProcessMonitor (Chapter 9), and Yap (Chapter 10)—provide additional insights into the workings of your program. ProcessMonitor lets you examine various characteristics of any process's activities: memory use, PostScript graphic states, the run-time environment, and so on. MallocDebug measures the dynamic memory use of an application. Yap lets you enter, edit, and execute PostScript code on the fly and allows you to read and write text files so the code can be used elsewhere.

Two tools are available to track off-screen drawing, which may affect what you see—or don't see—on-screen. The **NXShowPS** argument writes all PostScript code and values from the PostScript interpreter to the standard error stream. The **NXShowAllWindows** argument displays all of an application's windows, including those generated for off-screen imaging. Both of these are command-line arguments. To use them, start your program from a Terminal shell. On the command line, enter the program name followed by the parameter. For example

```
   /me/MyApps/NewApp/NewApp.app -NXShowAllWindows
```

starts the application **NewApp.app**, displaying all its windows as it runs.

Add Help to Your Application

Using Project Builder, Interface Builder, and Edit, you can create context sensitive help for your application. The standard help template provided by Interface Builder includes general information on the NeXTSTEP environment. You can add to this template to include application-specific help, and you can create links between the controls in your application and the help system to provide the user with context-specific assistance.

Translate Your User Interface

When the application is complete and help is available, you can create alternate versions with translated text for windows, panels, menu items, and buttons, as well as any help information you've added. NeXTSTEP application programming interface (API) provides ways of accessing bundles in your application containing the text and user interface in various languages you wish to support. "The Project Builder Application," Chapter 2, provides information on how to make a project localizable.

Make Your Application Available to Users

Once an application is debugged, you can install it in an application directory using Project Builder. Project Builder lets you determine which directory to install the application in and provides a way to automatically install the application when you build it.

When the user double-clicks a document file, the Workspace Manager has to locate and start the executable file for that application. Workspace Manager looks for the executable file in a systematic sequence of directory paths. This search sequence is contained in an environmental variable **path**. You can place an application in any of the directories specified in **path**.

Because of the search sequence specified by **path**, you can replace an application located later in the sequence with one of the same name earlier in the sequence. For example, **$(HOME)/Apps** occurs before **/NextApps** in **path**; if you place an application in the directory **$(HOME)/Apps** with the same name as an application in the **/NextApps** directory, the Workspace Manager finds and starts the version in **$(HOME)/Apps** (the Apps subdirectory in your home directory). You should consider the path when naming and installing applications.

If your application is intended for distribution on multiple floppy disks, you should configure it so that a user can install it using the Installer application. Tools for doing so are documented in **/NextLibrary/Documentation/NextDev/Concepts/Installer.rtf**.

2 The Project Builder Application

2 The Project Builder Application

Project Builder is the hub of application development in NeXTSTEP. It manages the components of your application and gives you access to the other development tools you use to create and modify these components. Project Builder is involved in all stages of the development process, from providing you with the basic building blocks for a new application to installing the application when it's finished.

Project Builder's unit of organization is the project. A project can be defined in two ways: conceptually and physically. Conceptually, a project comprises a number of source components and is intended to produce a given end product, such as an application. (Other types of end products are possible, as described below.) Physically, a project is a directory containing source files and Project Builder's controlling file, **PB.project**. This file records the components of the project, the intended end product, and other information. For a file to be part of a project, it must reside in the project directory and be recorded in the project's **PB.project** file. You don't edit **PB.project** directly; your actions in the Project Builder application—adding source files, modifying the project name or installation directory, and so on—have the effect of updating this file.

Project Builder can be used to create and maintain the following standard types of NeXTSTEP projects:

Type of Project	**Description**
application	A stand-alone NeXTSTEP application, such as those found in **/NextApps** or **/LocalApps**.
subproject	A project within a project. With larger applications, it's often convenient to group components into subprojects, which can be built independently from the main project. In building a project, Project Builder builds the subprojects as needed and then uses their end products—usually ".o" files—to build the main project.
bundle	A directory containing resources that can be used by one or more application. These resources might include such things as images, sounds, character strings, nib files, or executable code. For more information, see the class specification for the NXBundle class in *NeXTSTEP General Reference*. A bundle can be a stand-alone project, or contained within another project.
palette	A loadable palette that can be added to Interface Builder's Palettes window. See "Adding Custom Palettes, Inspectors, and Editors" in the next chapter for more information.

Project Builder also helps you prepare your application (or other type of project) for various language markets, a process called "localization". It does this by helping you group language-dependent components of your application—TIFF and nib files, for example—in subdirectories of the project. These subdirectories are named for a language and have a ".lproj" extension (for example, **Spanish.lproj**), and so are commonly called ".lproj" directories. Through the facilities of the NXBundle class, your application can load the appropriate, language-dependent components depending on the user's preferred language. (See the NXBundle class specification in *NeXTSTEP General Reference* and the file **/NextLibrary/Documentation/NextDev/Concepts/Localization.rtfd** for more information.)

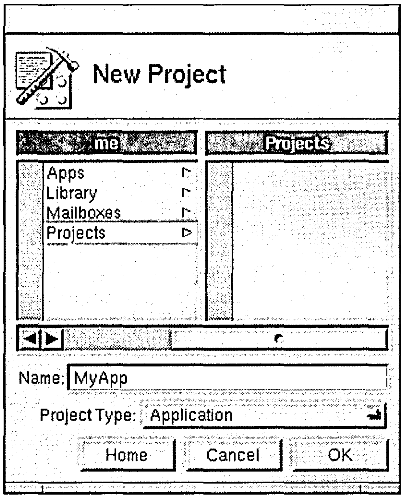
You can start Project Builder (located in **/NextDeveloper/Apps**) from the workspace as you would any other application, by double-clicking its icon in the workspace. When it starts up, only the main menu is visible. Once Project Builder is running, you can create a new project or open an existing project as described below.

Creating and Maintaining Projects in Project Builder

This section describes how to create a new project in Project Builder and how to convert a pre-3.0 project to the 3.0 project format. You'll also find information here about maintaining your project.

Creating a New Project

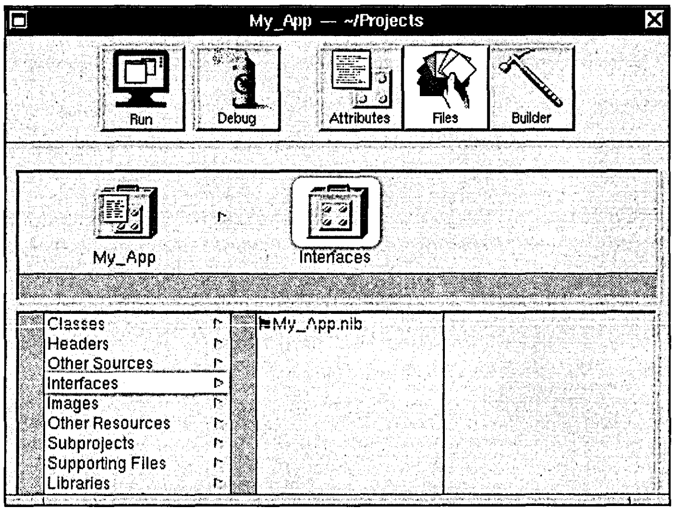
To create a new project, choose the New command in the Project menu. A panel appears in which you specify a pathname and name for the project. Specify a new directory on the Name line, or choose an existing directory in the browser (and leave the name **PB.project** in the Name field) if you want to use that directory as the root of the new project.



Name: MyApp

Project Type: Application

By default, the new project is a stand-alone application. A pop-up list in the panel lets you create a bundle or a palette instead. No matter what type of project you create, a project window for the new project appears.



You'll use this project window to maintain, build, and debug the project, as described in the rest of this chapter. For now, note the three modes of operation indicated by the three buttons in the upper right portion of the panel:

Mode	**Purpose**
Attributes	Set attributes of your project.
Files	Add, remove, or open project files.
Builder	Build the project.

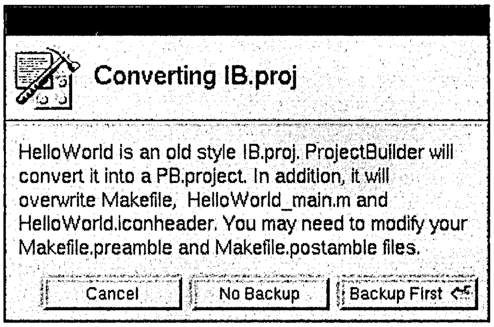
Opening an Existing Project

To open an existing project, choose the Open command in the Project menu. A standard Open panel appears in which you specify the project to open. Select the file named **PB.project** in the project directory and click Open to open the project.

When you open a project, its project window appears in Project Manager.

Opening and Converting a Pre-3.0 Project

To open an existing project that hasn't been converted to the 3.0 project format, choose the Open command in the Project menu. A standard Open panel appears in which you specify the project to open. Select the file named **IB.proj** in the project directory and click Open to open the project.



Converting IB.proj

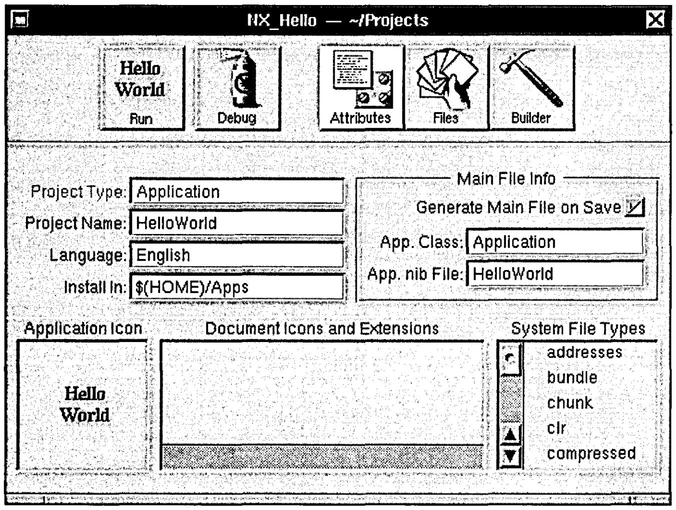
HelloWorld is an old style IB.proj. ProjectBuilder will convert it into a PB.project. In addition, it will overwrite Makefile, HelloWorld_main.m and HelloWorld.iconheader. You may need to modify your Makefile.preamble and Makefile.postamble files.

A panel appears warning you that the project file is an "old style IB.proj" which needs to be converted to a **PB.project**. (**Note**: Be sure to convert the project if you'll be continuing to maintain it in Release 3.0.) Since the conversion process overwrite several project files, you're asked if you want to back up those files first before converting the project. Unless you're sure you don't need to do this, you should click Backup First (or Cancel if you decide not to continue)—this causes a copy of the entire project directory to be made, with the name **CopyOf***ProjectDirectory*.

Once the project is converted, its project window appears in Project Manager. When you save the resulting project, it will be saved as a **PB.project** file in the same directory. This is the file you'll open in the future when you work with the project.

Setting Project Attributes

To bring up the Attributes display, click the Attributes button in the project window.

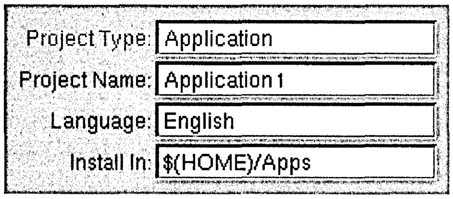


Project Type: Application
Project Name: HelloWorld
Language: English
Install In: $(HOME)/Apps
App. Class: Application
App. nib File: HelloWorld

The contents of the Attributes display varies depending on the type of project—application, bundle, or palette. The contents of these three types of Attributes display are shown below.

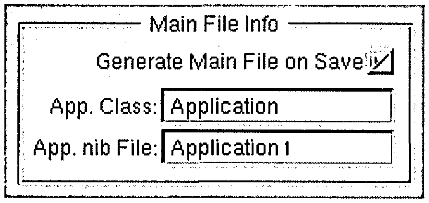
Application Attributes

If the project is an application, the Attributes display contains the following controls for defining application attributes.



Project Type: Application
Project Name: Application1
Language: English
Install In: $(HOME)/Apps

This group of controls includes fields for specifying the project name, the primary language (that is, the language in which the project is being developed), and the target directory.

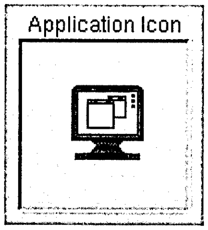


Generate Main File on Save ✓

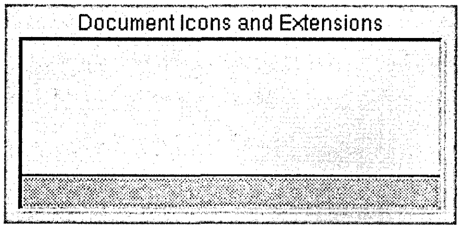
App. Class: Application

App. nib File: Application1

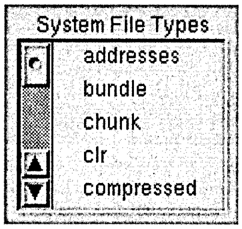
This group of controls includes fields for specifying the application class and the application's main nib file, plus an option for regenerating the Main file whenever you save the project. (Project Builder maintains this file and you aren't expected to change it; therefore you should leave this option checked, unless there's a reason why you need to maintain the Main file yourself)



The Application Icon well displays the application icon. The default application (shown here) is used if you don't provide one of your own choosing. To associate a new icon with the application, drag its TIFF file from the workspace into the well. The file is copied to the project directory, although it doesn't appear in any of the categories shown in the File display.



The Document Icons and Extensions well is where you indicate what types of documents your application is able to deal with. If you're creating your own document type, create a document icon for it and drag the TIFF file containing that icon into the well. Once the icon is in the well, change its label to match the document extension.

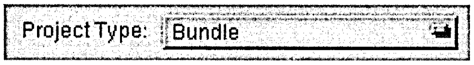

System File Types

- addresses
- bundle
- chunk
- clr
- compressed

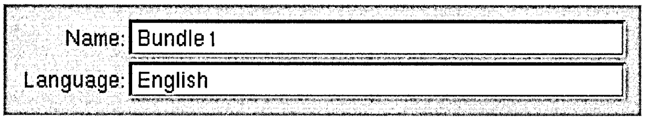
System File Types lists NeXTSTEP file types (as identified by their standard NeXTSTEP file extensions), any of which you may choose to have your application handle by selecting the file type in the scrolling list. When you select a file type by clicking it, a check mark appears next to its name, and it gets added to the Document Icons and Extensions well. Click the file type again if you want to deselect it and remove it from the well.

Bundle Attributes

If the project is a bundle (or subproject), the Attributes display contains the following controls for defining project attributes.


Project Type: Bundle

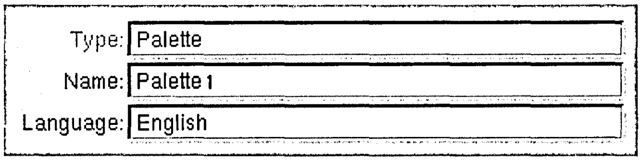
This pop-up list contains a Subproject item that lets you convert the bundle to a subproject. Note, however, that this is possible only with a bundle that's part of another project, not with a stand-alone bundle.


Name: Bundle1
Language: English

This group of controls includes fields for changing the project name and the primary language.

Palette Attributes

If the project is a palette, the Attributes display contains the following controls for defining project attributes.



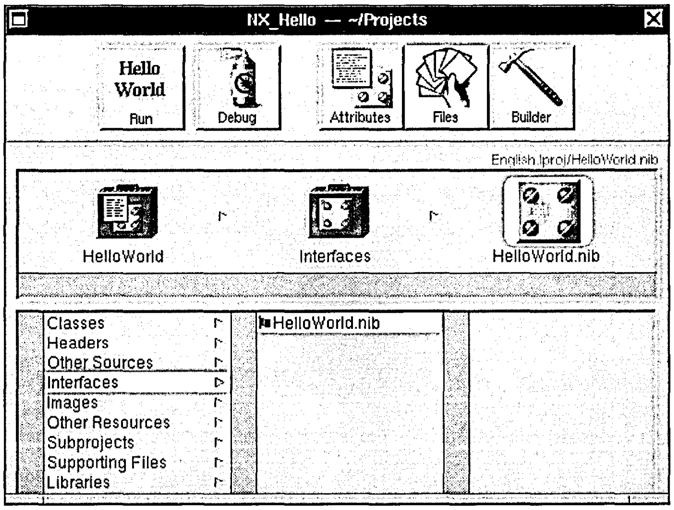
Type:	Palette
Name:	Palette1
Language:	English

This group of controls includes fields for changing the project name and the primary language.

Managing Project Files

The Files display of the project window is used to manage the files in the project. You can use this display to add or delete project files, as well as open them for viewing or editing.

To bring up the Files display, click the Files button in the project window.



The Files display provides a file viewer similar to the Workspace Manager's File Viewer, with categories of project components displayed in the left-hand column and project files for each category displayed to the right. Note that these project categories don't correspond to project subdirectories—the categories are logical rather than physical groupings of files.

The project directory provides you and Project Builder with a convenient way to organize the files used in putting together your application. As shown here, files in the project directory are grouped by Project Manager into a number of categories. These categories are represented with a suitcase icon (and are frequently referred to as "suitcases"). Briefly, these categories are:

Category	**Description**
Classes	Files containing code for custom classes used by an application.
Headers	Files containing declarations of methods and functions used by an application
Other Sources	Files containing code (other than class code) for an application. These may include ".m" files (containing Objective C code), ".c" files (containing standard C code), ".psw" files (containing PostScript code), and other sources. Project Builder automatically adds the file *ApplicationName*_**main.m** to Other Sources.
Interfaces	Nib files for each application and for each new module added to an application. The flag icon next to a file name in the Interfaces suitcase indicates that the file is localizable (that is, the file is in the *Language*.**lproj** subdirectory in the project directory, rather than in the project directory itself).
Images	Files containing images (other than icons) used by an application, including TIFF or EPS files.
Other Resources	Files (such as sound files) for other resources used by an application.
Subprojects	Directories containing subprojects used by an application
Supporting Files	Files not used directly by the application but that should be kept with the application.
Libraries	Libraries referenced by an application. NeXTSTEP libraries (including the default entries libNeXT_s and libMedia_s) are referenced but not copied into the project directory. Other libraries, such as those you create, may be added to the project directory.

You can use Project Builder's file viewer to:

- Browse the project and the files it contains.
- Add files to the project (as described below).
- Remove files from the project by selecting the file in the browser and then choosing Remove in the Files menu.
- Open a project file by double-clicking its name or icon (or, selecting the file in the browser and then choosing Open in Workspace in the Files menu).

There are in fact several ways to add an existing file to a project. The file can be already located in the project directory, or it can be somewhere else. To add it, use one of the following methods:

- Drag the file from the File Viewer into the project window. If you drag it to the suitcase it belongs in, that suitcase will open up. If you let it go, it will be added to that suitcase. If instead you drag it to the project suitcase, the project suitcase will open up and the file will be added to it. The Classes suitcase takes ".m" files, the Headers takes ".h" files, and so on. "Other Sources" refers to files that are not headers or classes, but need to be compiled and linked into the target of the project (application, bundle or palette). "Other Resources" refers to files that need to be copied into the target. "Supporting Files" refers to files that are necessary to maintain the project, but don't end up in the target.
- Select a suitcase and choose the Add command in the Files menu (or simply double-click the suitcase). A panel will appear, in which you specify a file to add to the selected suitcase.
- Use the service that Project Builder supplies to other applications. Relevant applications have a command named Project in their Services menu. This command brings up a submenu containing two commands: Add To and Build. Add To can be used to add the current file to the project (in this case, the file must already be located in the project directory).

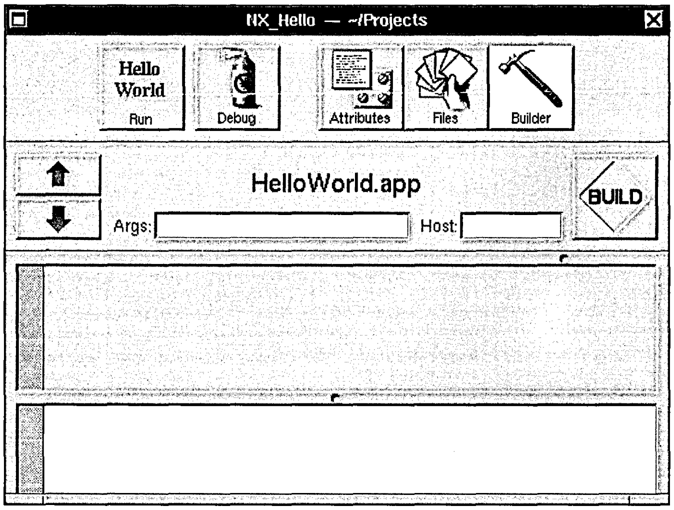
Also note the following shortcuts available in the File display:

- Control-dragging in a file list allows you to reorder the files. This can be especially important in dealing with libraries, since the file order determines the link order.
- Alternate-double-clicking the icon of a selected file selects that file in the workspace File Viewer, instead of opening it.
- Command-double-clicking a source file opens both the file and its associated header file, if it exists.

Building the Project

When you instruct Project Builder to build the project, the project is compiled by the **make** program using the project's makefile. The project's source files are compiled and linked into an executable file. The project makefile provides the information **make** needs to do this job. The warnings generated by the compiler and link editor provide information to help you locate and fix bugs detected at compile time.

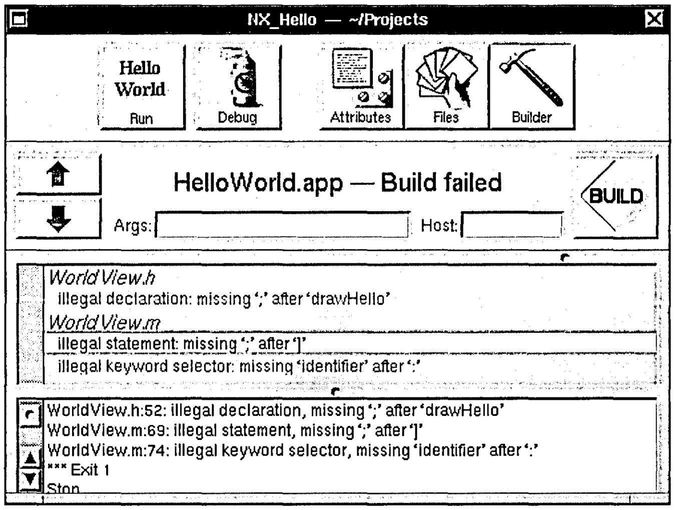
To build the project, first bring up the Builder display by clicking the Builder button in the project window.



The Args field is for specifying build arguments to be passed to **make**; the Host field is for specifying a remote host machine on which to build the project. Leave these fields blank if you don't have anything to specify. If you want to specify **make** arguments or a host name, be sure to do so first before starting to build the project.

Note: If you build the project on a remote host, be sure you know what version of NeXTSTEP the host is running.

When you're ready to build the project, click the Build button. As the build progresses, the two views at the bottom of the window inform you of any warnings or error messages that occur—the upper Summary view is more selective in what it chooses to display, so you may choose to hide the lower Detail view and only refer to its output when you need to.



HelloWorld.app — Build failed

```
WorldView.h
  illegal declaration: missing ';' after 'drawHello'
WorldView.m
  illegal statement: missing ';' after ']'
  illegal keyword selector: missing 'identifier' after ':'
```

```
WorldView.h:52: illegal declaration, missing ';' after 'drawHello'
WorldView.m:69: illegal statement, missing ';' after ']'
WorldView.m:74: illegal keyword selector, missing 'identifier' after ':'
*** Exit 1
Stop
```

If an error is encountered during the build process, a message appears in both the Summary view and the Detail view, as shown here. Click a line in the Summary view to open the specified file; if you click a line containing an error message (shown in red on color displays and bold on monochrome displays), the file opens in Edit and scrolls to display the line that contains the error.

Build Targets

app.make (the shared makefile used to generate the executable file for all applications created with Project Builder) defines a number of alternate targets to perform specific tasks at various phases of the application development process. To run **make** using the alternate targets, enter the corresponding argument in the Args text field of the Builder display.

The following table lists various targets and the tasks they perform.

Target	**Task**
none	If no target is specified, compiles and links a debuggable, optimized version of the executable file. This is the default target used when you give the Build command without an argument.
debug	Compiles (with all warnings and -DDEBUG on) and links a debuggable, unoptimized version of the executable file with the extension ".debug".
clean	Removes all derived files, such as object and executable files, from the project directory, returning the project to its precompiled state.
install	Builds (if needed) and copies the application into the installation directory specified in Project Builder, setting permissions and owners as appropriate. The default is **$(HOME)/Apps**, the **Apps** directory in the user's home directory.
installsrc	Installs the source files for the project into the directory specified in the SRCROOT variable in a command-line argument (you must specify the target directory on the command line). If the directory exists it (and its contents) will be deleted, and then be recreated before the source files are moved there. This option is useful for archiving completed projects.
depend	Generates an optional **Makefile.dependencies** file, containing a complete dependency graph for the project, including headers. Once this file exists in the project directory, it's conditionally included by your project makefile.
profile	Generates (with all warnings and -DPROFILE on) the file *ApplicationName***.profile**, an executable containing code to generate a **gprof** report. This option is useful when you are performance tuning an application. See the UNIX manual page **gprof** for details on profiling.
help	Lists these targets with their parameters.

The Preamble File

Sometimes it's necessary to alter the standard build process as defined by the project makefile. You do this by adding to the project a **Makefile.preamble** file that overrides the macros defined in the project makefile. To override a macro definition in the project makefile, include a definition for the same macro in **Makefile.preamble**. For example, the following definition for the macro INSTALLDIR always appears in the project makefile:

```
INSTALLDIR = $(HOME)/Apps
```

This macro causes the **make** install target to place the executable in the Apps subdirectory of your home directory. To have install place the executable in another directory, define the following macro in **Makefile.preamble**:

```
INSTALLDIR = /LocalApps
```

To use one of the macros listed above in **app.make**, you first define it in **Makefile.preamble**. You can, for example, define link editor flags to add segments to your executable file. For example, an application might defines the following macro in its **Makefile.preamble**:

```
LDFLAGS = -segcreate EXTRA document extra.rtf
```

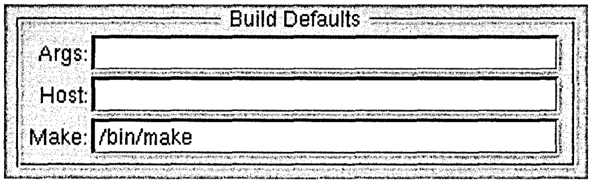
Using this macro definition, the link editor will create a segment named "EXTRA" in the executable file; that segment will have a section named "document" containing the document file **extra.rtf**.

See the makefiles in **/NextDeveloper/Makefiles** for more information.

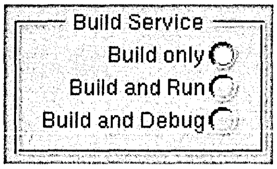
Setting Preferences

You can specify preferences for a variety of options using the Preferences panel. To bring up the panel, choose the Preferences command in the Info menu.

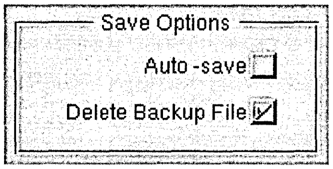
Enter values or click buttons to specify new preferences, as described below. Then click Set to set the new preferences (or click Revert to restore the previous settings). Note that the settings on the Preferences panel are global—they apply to all projects, not just the current project.



The controls in the Build Defaults group let you specify build arguments to be passed to **make**, a remote host on which to build the project, and an alternative to **/bin/make**, the standard **make** program.



The controls in the Build Service group let you specify what (if anything) you want to have happen after building your project (specifically, after building your project by choosing Project Builder's Build command on the Services menu)—Build only, Build and Run, or Build and Debug.



Auto -save [ ]
Delete Backup File [✓]

The controls in the Save Options group let you specify whether projects should be auto-saved, and whether the most recent backup file is automatically deleted or retained.

Running and Debugging an Application

In addition to maintaining and building a project, you can use Project Builder to run or debug the resulting application, as described in this section.

Running

To run the project application, click the Run button in the project window. If the project hasn't been built yet, it's built and then the application is run. The Run button's icon is the same as the application icon—the icon shown here is the default application icon that's used if no other icon is specified in the Attributes display.

Tip: Alternate-clicking the Run button runs the application without building it first.

Debugging

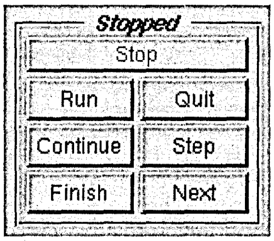
To debug the project application, click the Debug button in the project window. If the project hasn't been built yet, it's built first and then the application is run in debug mode.

Tip: Alternate-clicking the Debug button runs the application under the debugger without building it first.

When you indicate that you want to debug an application in Project Builder, the following steps occur:

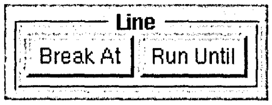
- The project is built (unless it's already up to date).
- Terminal creates a new window to run the GDB process in.
- As GDB starts, it's instructed to read the **PB.gdbinit** file in the project directory.
- The **view** command in the **PB.gdbinit** file is executed and causes a command named Gdb to appear in Edit's main menu.

Choose the Gdb command from Edit's main menu to display the GDB control panel. This panel has the application name as its title, and contains four groups of controls for interacting with GDB as you debug the application. (GDB commands that aren't accessible through the panel can still be executed manually in the shell window in which GDB is running.)

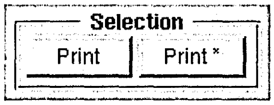


The first group (labeled either *Running* or *Stopped*) contains the following buttons for controlling the execution of the application.

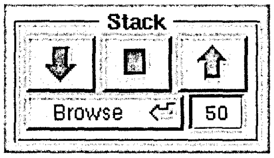
Button	**Description**
Run	Starts the application being debugged.
Continue	Continues the application being debugged, after a signal or breakpoint.
Finish	Executes until the selected stack frame returns. (Upon return, the returned value is printed and put in the value history.)
Quit	Exits GDB.
Step	Steps the application until it reaches a different source line.
Next	Steps the application, proceeding through subroutine calls. The Next command is like the Step command as long as there are no subroutine calls; if there are, the call is treated as one instruction.



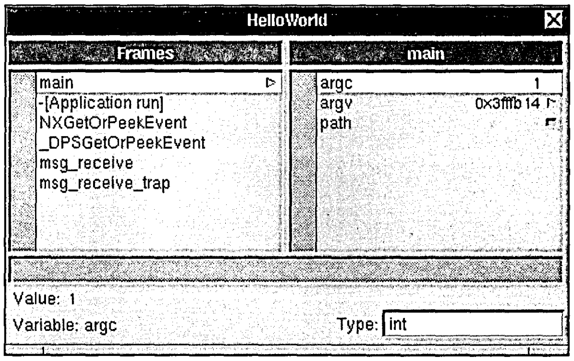
The Line group contains controls for setting breakpoints in source files and running until a breakpoint is reached. These controls use Edit's current file and line as their argument. Click the Break At button to set a breakpoint at the line containing the insertion point in the main Edit window. Click the Run Until button to run the application until it reaches the next breakpoint.



The Selection group contains controls for evaluating and printing the value of a C or Objective C expression. These controls use Edit's current selection as their argument. Click the Print button to display the value of the selected expression. Click the Print* button to display the value that the expression points to.



The Stack group contains controls for browsing the data in the program being debugged. Clicking the Browse button causes the following browser panel to appear:



Frames	main	
main	argc	1
-[Application run]	argv	0x3fffb14
NXGetOrPeekEvent	path	
_DPSGetOrPeekEvent		
msg_receive		
msg_receive_trap		

Value: 1
Variable: argc Type: int

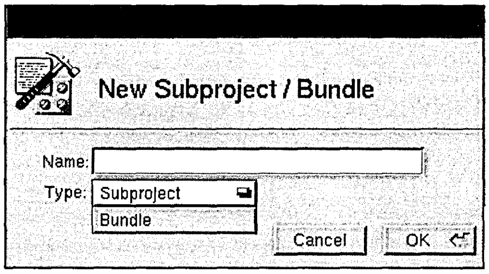
You can use this browser to select and inspect particular stack frames and their variables.

Project Builder Command Reference

Project Builder's main menu contains the standard Info, Edit, Windows, Services, Hide, and Quit commands. All commands unique to Project Builder are located in the Project and Files submenus—these menus and the commands they contain are described below.

Commands in the Project Menu

The Project menu contains commands for creating and maintaining your projects.

Command	Description
New	Creates a new project.
Open	Opens an existing project.
Open Makefile	Opens a window for just the Makefile of a project and displays the Builder view in the window. To build the project, click Build.
Save	Saves the current project.
New Subproject	Creates a new subproject. A panel appears in which you specify the name and type of subproject. The type can be either Subproject or Bundle.  Specify a name and type, and then click OK to add the subproject or bundle to the current project.
Add Help Directory	Adds a Help directory to the current project. A template Table of Contents file and Index file are placed in the Help directory. For more information on adding help to an application, see Chapter 3.

Run Application	Runs the application associated with the project, just as if you had clicked the Run button in the project window.
Debug Application	Debugs the application associated with the project, just as if you had clicked the Debug button in the project window.
Build Application	Builds the application associated with the project, just as if you had clicked the Build button in the project window.

Commands in the Files Menu

The Files menu contains commands that affect the files that make up a particular project. Commands in this menu are enabled only when the Files view for the project is selected.

Command	**Description**
Add	Adds a file to the selected suitcase in the current project. Be sure to select the appropriate suitcase in the File view before choosing the command.
Open in Workspace	Opens the selected file in the application that's registered with the Workspace Manager as the default application for files of that type.
Select in Workspace	Displays and highlights the selected file in the Workspace Manager's File Viewer window.
Remove	Removes the selected file from the current project (without deleting it from the project directory).
Sort	Alphabetically sorts the files in the current suitcase.
Make Global	Makes the selected file global (that is, moves it from the *Language*.**lproj** directory into the project directory).
Make Localizable	Makes the selected file localizable (that is, moves it from the project directory into the *Language*.**lproj** directory).

3

The Interface Builder Application

3 The Interface Builder Application

Interface Builder is a tool that helps you design and build applications. It speeds the creation of applications by letting you define an interface (and in some cases, an entire application) graphically rather than by writing C and Objective C code. With Interface Builder, you drag objects from palettes of NeXTSTEP objects directly into the application you're building. Once there, an object can be modified in ways that are specific to its class: You can set a Button object's title or set the minimum and maximum values of a Slider object, for example. After you've gathered and edited the objects that will make up your application, Interface Builder lets you define the interactions among them and associate help messages with each of them. Even before you write a line of code, you can run your application within Interface Builder to check the operation of its interface.

Interface Builder's technique of direct manipulation of programming objects isn't limited to objects defined in NeXTSTEP. Interface Builder's palettes are extensible, letting you load palettes containing objects that you or other developers have created.

In many ways, using Interface Builder to create an application is much like using a graphics editor to create a drawing. However, Interface Builder is not a simple "screen painter" or form-generation tool. When you build an application with Interface Builder, you are interacting with the actual programming code that will be executed when your application runs on its own. The objects you manipulate in Interface Builder are the objects that will appear in the working version of your application. If your application runs correctly in Interface Builder, it will run correctly on its own.

The work you do in Interface Builder is saved in a *nib file* (a file package having a name ending in ".nib", which stands for "NeXTSTEP Interface Builder"). This file contains archived versions of the objects you assembled for your application, information about connections between these objects, and other information. When an application begins running, it unarchives these objects and associated information from one or more nib files. Nib files are discussed in more detail late in this chapter, but for now it's important to note that projects in NeXTSTEP contain at least one nib file and that Interface Builder lets you create and modify these nib files.

As pointed out in the previous two chapters, the central tool for developing applications in NeXTSTEP is Project Builder. When you start a new project in Project Builder, you're provided with several standard components, one being a nib file. When you want to modify this standard nib file, Project Builder invokes Interface Builder as the nib file's editor. Interface Builder and Project Builder are interlinked in other ways as well. As you define new classes, import images or sounds, or create new nib files, Interface Builder and Project Builder work together to keep each other aware of the state of the project.

Even if you're new to this computing environment, you'll find that with Project Builder and Interface Builder, you'll be able to create applications 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 application development will be for you. Thus, we recommend that you familiarize yourself with the material in *NeXTSTEP Object-Oriented Programming and the Objective C Language* and at least scan the class specifications (located in Chapter 2 of the *NeXTSTEP General Reference* manual) for the major Application Kit classes before attempting to take your work with these tools beyond the experimental stage.

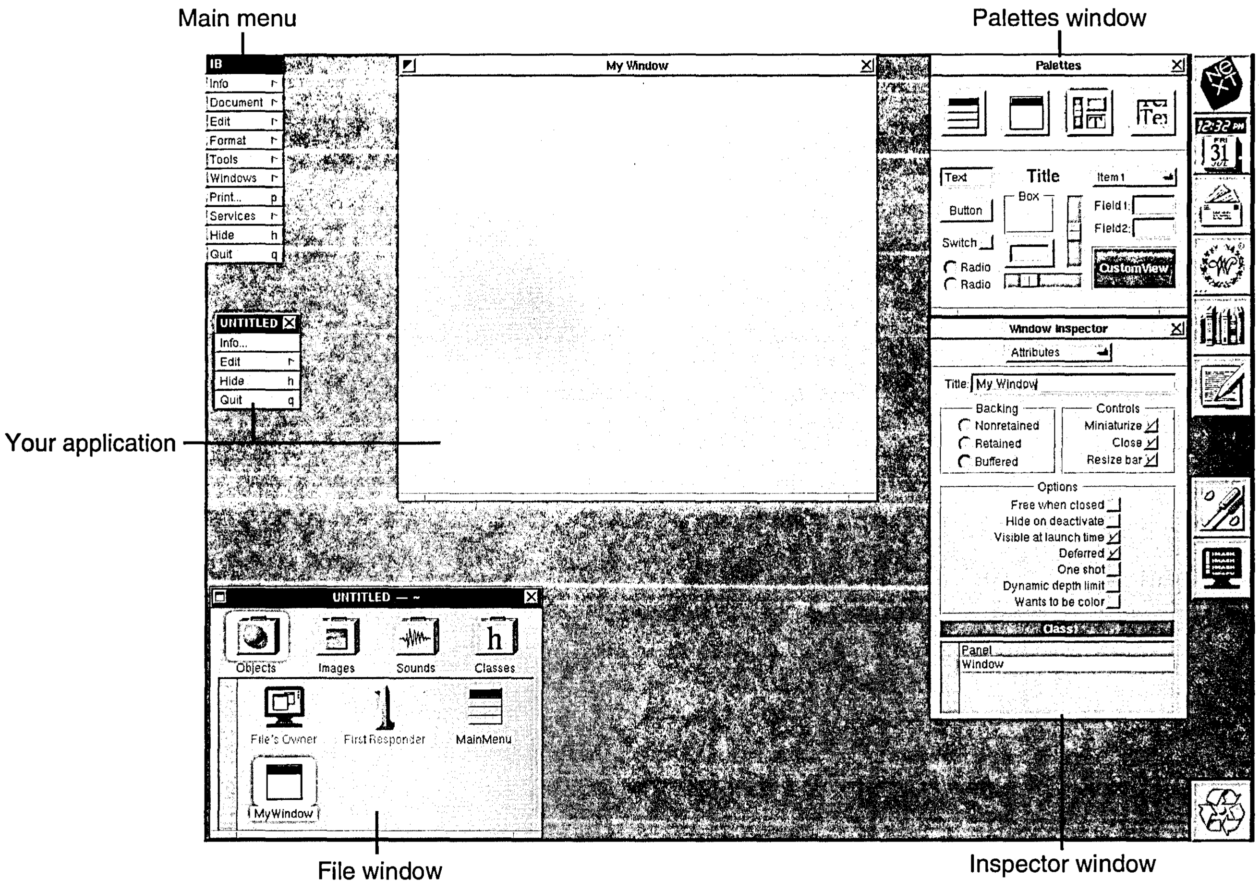
This chapter provides general reference information on Interface Builder. It first introduces Interface Builder's major components and then discusses some of the common tasks that you use Interface Builder to accomplish. A final section provides a quick reference for each of Interface Builder's commands.

For a tutorial-based introduction to this tool—and to programming in NeXTSTEP in general—see Chapters 15 through 18 of this manual. Interface Builder's application programming interface (API), which allows you to create custom palettes, is described in detail in Chapter 8 of the *NeXTSTEP General Reference* manual.

The Basics

An Orientation

When you use Interface Builder, its windows—and the windows of the application under construction—share the screen. The illustration gives you an idea how this looks.



Title: My Window

Backing: Nonretained, Retained, Buffered

Controls: Miniaturize ✓, Close ✓, Resize bar ✓

Options: Free when closed ☐, Hide on deactivate ☐, Visible at launch time ✓, Deferred ✓, One shot ☐, Dynamic depth limit ☐, Wants to be color ☐

Class: Panel, Window

Interface Builder's windows frame an area of the workspace where you build your application. At the upper left is the main menu, which gives you access to Interface Builder's tools and commands, and at the upper right is the Palettes window. The *Palettes window* is the source of objects (Buttons, Sliders, Windows, and so on) that you can drag into your application. Below the Palettes window is the *Inspector window.* You use this window to set the attributes of an object, to connect it to other objects, and to review the attachments between objects and help messages. At the bottom left is the *File window.*

The File window displays your application's top-level objects (its windows, main menu, and so on) and gives you access to the image, sound, and class resources that are available to your application.

Building an Application with Interface Builder

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 the Palettes window into 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 NeXTSTEP applications.

(Other NeXTSTEP software kits, such as the Database Kit™, can supply user interface objects in the form of Interface Builder palette objects. This discussion applies to those as well, but for simplicity this discussion focuses on the Application Kit since it provides the preponderance of user-interface objects.)

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 the Inspector window that lets you set the values for particular 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. For example, a button can be connected to the window it appears in so that when the button is clicked, the window closes. Such connections are made through an object's outlets. An *outlet* is an instance variable that identifies another object in the application. Common examples of outlets include a Control's **target** or an Application or Window object's **delegate**.

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 helps you declare classes that encapsulate the code that's unique to your application.

Interface Builder and the Objective C language encourage a style of programming that puts your application's unique code in one or more objects of your own design. The application's user-interface objects handle routine business, such as displaying the main menu 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 Nib File

The work you do in Interface Builder is saved in a *nib file*—actually, a file package whose name ends in ".nib". This file contains:

- Archived objects. The Buttons, NXBrowsers, TextFields, and other objects that you dragged into your application's windows while designing your application's user-interface are archived in the nib file. The archived information includes the object's class and other attributes, such as its size, location, and position in the view hierarchy (for View objects).
- Class interface information for any subclasses that you define. At run time, the Application Kit sends messages to create objects of these classes.
- 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 nib file's owner is described in the next section.)

NeXTSTEP applications have at least one nib file, the *main nib file*. This file contains the specifications for the application's main menu and perhaps other objects. An application can have only one main nib file.

More complex applications may have other nib files in addition to the main nib file. For example, an application might have two nib files, one containing the archive for the main menu and other primary interface objects and the other containing an archive for a Find panel. If a user issues a search command, the Find panel is created by loading the objects from the secondary nib file. Since the Find panel isn't created unless it's needed, the Find panel's objects don't consume system memory or add to the application's start-up time.

Interface Builder's File window gives you a summary view of the contents of a nib file. Each window in the nib file is represented by a window icon in the File window. By double-clicking a window icon, you can bring the window it represents to the front so that the objects it contains are visible. The File window also contains icons that represent the file's owner object and a "First Responder" object, objects that are discussed in the next sections.

The Nib File's Owner

The nib file's owner is an object that's external to the nib file and that is the conduit for messages between the objects that will be unarchived from the nib file at run time and the other objects in your application. In general, the core objects in your application access the objects unarchived from the nib file indirectly through owner object. In turn, the unarchived objects communicate with the other objects in your application by sending messages to the owner object.

Each nib 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 nib file. The owner may also have outlets that will be initialized at run time to point to the objects within the nib file.

The owner must exist before the interface objects are loaded. For example, Project Builder generates a main file that follows this sequence of messaging to create the owner, load the interface information, and then run:

```
[Application new];
if ([NXApp loadNibSection:"HelloWorld.nib" owner:NXApp withNames:NO])
    [NXApp run];
```

Note: NXApp is a global variable that identifies the Application object, the object that's created by the message in the first line of the example above. (For more information on the **loadNibSection:owner:withNames:** method—and especially on the search path it uses for locating the appropriate nib file to load—see the specification for the Application class.)

What happens when the nib file is loaded at run time is described in "The Nib File at Run Time," later in this chapter.

The First Responder Object

The First Responder icon in the File window 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 target.

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. In this respect, the First Responder icon is really a fiction since it identifies no one particular object, but rather any object having a particular status. This fiction, however, is very useful.

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. If you create a new application in Project Builder, open its nib file, and check the connections in the application's Edit menu, you'll discover that all Edit commands are connected to the First Responder.

(Incidentally, the ability to let the target of a message be defined at run time rather than at compile time is an example of dynamic binding in Objective C. For more information on Objective C, see *NeXTSTEP Object-Oriented Programming and the Objective C Language*.)

The Nib File at Run Time

As pointed out previously, the standard main file generated by Project Builder includes these messages:

```
[Application new];
if ([NXApp loadNibSection:"HelloWorld.nib" owner:NXApp withNames:NO])
    [NXApp run];
```

The **loadNibSection:owner:withNames:** messages invokes code within the Application Kit that unarchives the nib file's objects, instantiates custom objects, establishes connections between objects, and finally informs these new objects that they have been loaded. The following sections describe these steps in more detail.

Step 1: Unarchiving Objects

The first step the system takes in loading a nib file is to unarchive the objects it contains. An object archive records the class and salient data structure of a particular object. To unarchive an object, the system allocates memory for the object (by sending the class object an **alloc** message) and then sends the newly allocated object a **read:** message to read in the applicable data that was preserved in the nib file. For example, to unarchive a Button object, a new Button object is allocated and then sent a **read:** message to read information such as its title, size, identity of its superview, and so on from the nib file. As part of the unarchiving system, all objects receive an **awake** message after they have been unarchived. (See the Object class specification in the *NeXTSTEP General Reference* manual for more information on archiving.)

Step 2: Instantiating Custom Objects

A nib file can also contain references to objects that you have defined. (See "Defining New Classes" later in this chapter for more information.) For example, you could use the Classes display of the File window to define a subclass of Button called RepeatButton. (Presumably, when pressed, a RepeatButton sends its action message repeatedly at a given interval.) To add a RepeatButton to your application, you drag a CustomView object into your application's window, set its size and location, and reassign its class to be RepeatButton. When you save the nib file, Interface Builder records that an object of the RepeatButton class is of a certain size and location within your application's window. The object itself isn't archived since the code for the class isn't accessible to Interface Builder; in fact it may not yet exist!

As the nib file is loaded at run time, the system will attempt to instantiate this RepeatButton object. Assuming you have linked the code for the RepeatButton class into your application, the RepeatButton class object will receive an **alloc** message and then an **initFrame:** message, and the object will be instantiated.

The **initFrame:** message is used only for custom objects that inherit from View. For non-View objects, an **init** message is sent instead. (Custom objects that don't inherit from View are represented by a sphere icon in the File window.)

Note the distinction between the messages a custom object receives and the messages an archived object receives. When a nib file is loaded, a custom object receives an **init...** message but not a **read:** or **awake** message. Conversely, an unarchived object receives **read:** and **awake** messages but not an **init...** message. (However, objects of both types can receive **awakeFromNib** messages, as described later in this chapter.)

This distinction becomes important when you create custom palettes for objects that you have defined. (The section "Setting Preferences" discusses custom palettes.) For example, using a CustomView, you might add a RepeatButton to an application and find that the application operates properly when executed. Then, you might create a custom palette containing a RepeatButton object. An application built using a RepeatButton from this custom palette may not operate properly unless a RepeatButton's unarchiving methods can establish its state as completely as its **init...** method does.

Step 3: Establishing Connections

As the next step in loading the nib file, code in the Application Kit establishes connections between the objects that were created in the previous steps.

You make connections between objects in Interface Builder using the Connections inspector. (Connections are described in more detail in "Setting Connections" later in this chapter.) When you establish a connection, you identify a source object, an outlet of that object, and a destination object. For example, the source could be a Window object, the outlet could be the Window's **delegate** instance variable, and the destination object could be a custom object. By establishing this connection, you are specifying that, at run time, the custom object will be the Window's delegate, and thus capable of receiving any of the messages that a Window sends to its delegate. (As described later, some connections are more constrained in that they specify not only an outlet and a destination object, but a specific message to be sent to that object.)

After the nib file's archived objects are unarchived and its custom objects instantiated, the connections that were established in Interface Builder are reestablished with these run-time objects. Using the example above, the Window is unarchived, the custom object instantiated, and then the Window's **delegate** outlet is set to the **id** of the custom object.

Connections to and from the file's owner object are also established at this time. This is possible since the method that loads the nib file takes as one of its arguments the **id** of the file's owner:

```
[NXApp loadNibSection:"HelloWorld.nib" owner:NXApp withNames:NO]
```

Connections between objects are established in one of two ways. If the source object responds to a **set*MyOutlet*:** message, it will be sent that message. So, using the example above, the Window object would receive a **setDelegate:** message. (Note that the system determines the message to send by capitalizing the first letter of the outlet's name and prepending "set".) If the object doesn't respond to such a message, the value of its outlet instance variable is set directly, without a message being sent. Thus, you don't have to implement a **set*MyOutlet*:** method for each outlet you declare for a custom object.

Step 4: Sending awakeFromNib Messages

As the last step in loading the nib file, the system sends **awakeFromNib** messages to the objects that were derived from the information in nib file. Any object that was created from the nib file can receive this message if it implements the corresponding method.

The **awakeFromNib** message signals that the loading process is complete. It's guaranteed that when an object receives an **awakeFromNib** message, all of the nib file's archived objects have been unarchived, all of its custom objects instantiated, and all connections recorded in the nib file established.

Using Interface Builder

Manipulating View Objects

In general, in Interface Builder you interact with View objects through direct mouse manipulation. To select an object in your application's window, click it. An object indicates that it's selected by displaying eight small squares—or *control points*—around its perimeter. You can select multiple objects by holding down the Shift key while clicking each one in turn. You can also select multiple objects in a window by "rubberbanding" them. That is, you position the cursor to one side of the objects, press the mouse button, and drag diagonally across the objects. Your motion with the mouse describes a rectangular area marked with a fine, dotted outline. When you release the mouse, any object contained in the area, or intersected by the outline, becomes selected.

Once an object is selected, you can resize it by dragging one of its control points. To reposition the object, drag anywhere within the object, but not on one of the control points.

Double-clicking an object selects some feature within it. For example, single-clicking a button selects it, but double-clicking the button moves the focus of selection to the button's title.

For more complex Views, double-clicking lets you move the focus of selection down the view hierarchy. For example, if you double-click a Box object that contains Button subviews, the focus of selection moves to the Button objects. Now that the focus in within the Box, double-clicking one of the Buttons selects the text within it.

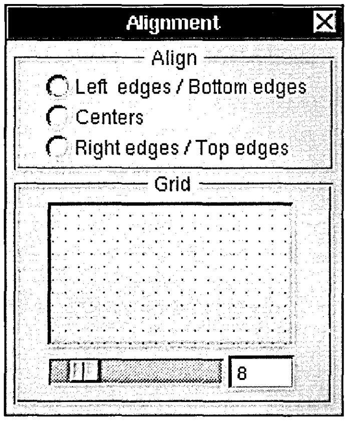
The mouse can also be used to resize the document view of a ScrollView. Imagine that you have selected one or more objects in a window and then issued the Group in ScrollView command. To manipulate these newly grouped objects, you double-click within the area of the ScrollView. Now, if you move the cursor to the top or right edge of the ScrollView, the cursor changes to a double headed arrow indicating that you can resize the ScrollView's document view. By pressing the mouse button and dragging away from the center of the ScrollView, you simultaneously increase the size of the document view and scroll the ScrollView to make the new portion of the document view visible. Scroll buttons and a scroll knob appear to indicate that the document view exceeds the dimensions of the ScrollView. Using this technique, you can specify which portion of the ScrollView will be visible at run time.

Using the Layout Commands

The commands in the Layout menu help you arrange objects in a window. You'll find it easiest to learn their operation through experimentation. For example, add two buttons to an application window, select both, and then experiment with the Layout commands. See the command descriptions at the end of this chapter for more information.

Using the Alignment Panel

This panel lets you set the spacing of the alignment grid, the grid that help you position objects accurately within a window. It also lets you set the reference point by which objects are aligned to the grid.



The three radio buttons let you set whether an object's lower left corner, center, or upper right corner are constrained to an intersection of the grid. The grid size field displays the value used for the vertical and horizontal spacing of the grid. You can adjust this value either by the slider or by entering a value in the grid size field. The value must be an integer in the range from 4 to 32. The area above the slider displays the current setting of the grid spacing.

Tip: Although in general it's best to leave the grid on and align your interface objects to it, at times, you may want to position an object off the grid for aesthetic reasons. To do this, turn the grid off, position the object, and then turn the grid back on. Interface Builder will leave the object where you put it despite the grid setting, as long as you don't try to move the object when the grid is on.

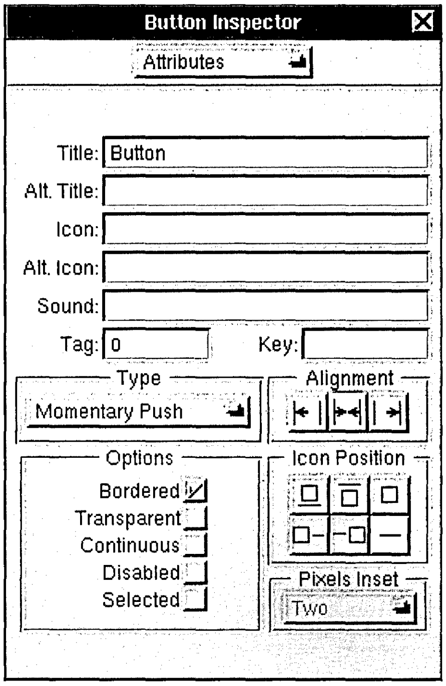
Inspecting Objects

You use the Inspector panel to edit the properties of both View and non-View objects in your application. The Inspector panel appears when you click the Inspector command in Interface Builder's Tools menu.

The panel has many personalities. Its contents are determined by Interface Builder's selection: If a Button object is selected, the Inspector panel displays the Button Inspector; if a Window is selected, the panel displays the Window inspector. (The Inspector panel's title announces the class of the selected object.) In addition, the panel itself has four displays—Attributes, Connections, Size, and Help—which are accessible through the pop-up list at the top of the panel. These four displays are discussed in the following sections.

Setting Attributes

The Attributes display of the Inspector panel lets you set the selected object's basic characteristics. For example, the illustration shows the Attributes display of the Button Inspector.



Conceptually, each of the characteristics in the Attributes display corresponds to an Objective C message that the selected object responds to. For example, this table shows the correspondence between some of the attributes displayed in the illustration above and messages that a Button object understands:

Attribute	**Message**
Title	setTitle:
Alt. Title	setAltTitle:
Icon	setImage:
Sound	setSound:
Icon Position	setIconPosition:

If you have questions about any of the attributes displayed for a selected object, you should consult the class specification (in *NeXTSTEP General Reference*) for that object.

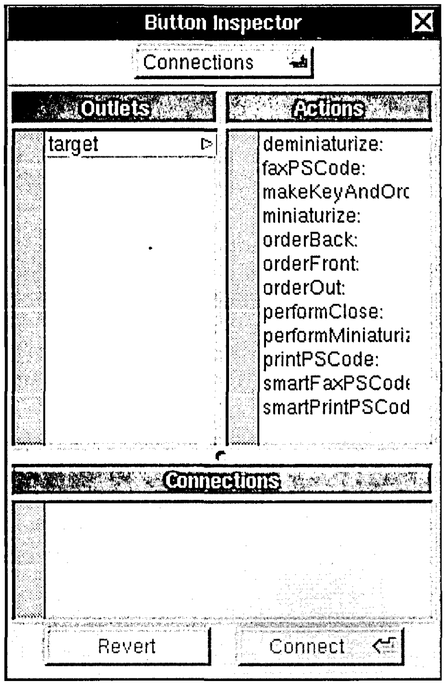
The Attributes display for the File's Owner, for CustomViews, and for custom objects that you instantiate in the File window lets you set the class of these objects.

The Attributes display for images in the File window's Images suitcase shows the image and its dimensions. This display doesn't allow you to edit the image or the dimensions. It's primarily used to examine bitmaps that are too large to be displayed in the Images display of the File window.

If you select a sound icon in the Sounds display of the File window, the Inspector panel displays the Sound inspector. This inspector lets you display, record, play, and make limited modifications to sounds. The inspector 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. 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—for editable sounds. The Record button starts recording through the microphone. 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.

Setting Connections

The Connections display of the Inspector panel lets you establish and review connections between objects. You create a connection by Control-dragging a line from the source object to the destination object. When the destination has been unambiguously identified, Interface Builder draws a rectangle around it. Releasing the mouse button completes the operation. If the Inspector panel 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 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.

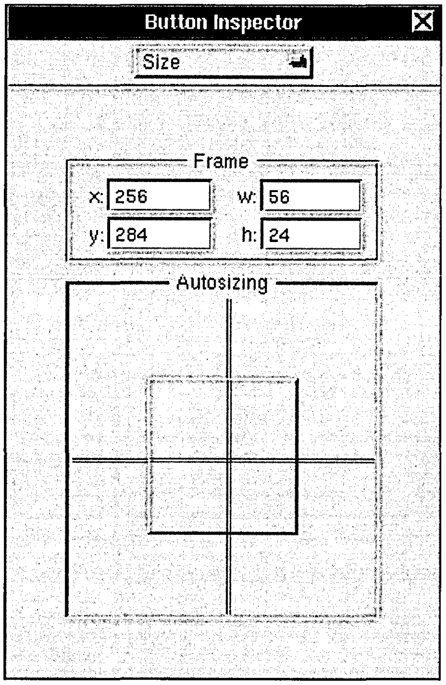
The left column of the display lists the outlets of the source. If the outlet is named "target" (in other words, the source object is a Control) the right column lists the action messages that the destination object can respond to. By selecting an outlet in the left column and, for sources that are Controls, selecting an action message in the right column and clicking the Connect button, you establish a connection between the source and destination objects. That connection is listed in the lower portion of the panel.

Tip: By clicking the entry in the Connections list, you can have Interface Builder display the connection line between the source and destination objects.

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.

Setting Size Characteristics

The Size display lets you set the precise dimensions and location of the selected object and specify how the object will respond to resizing.



x: 256 w: 56
y: 284 h: 24

The Frame fields let you set the object's location, width, and height. Normally, you set these attributes through direct mouse manipulation of the object. These fields are provided for those situations in which more precision is required.

The Autosizing portion of the display presents a schematic of the selected object and its surroundings. The small square represents the selected object and the area around it represents its superview or other surroundings. (If the selected object is a window, the small square is replaced with a window image.) The horizontal and vertical lines that bisect the squares are the controls that let you set resizing behavior.

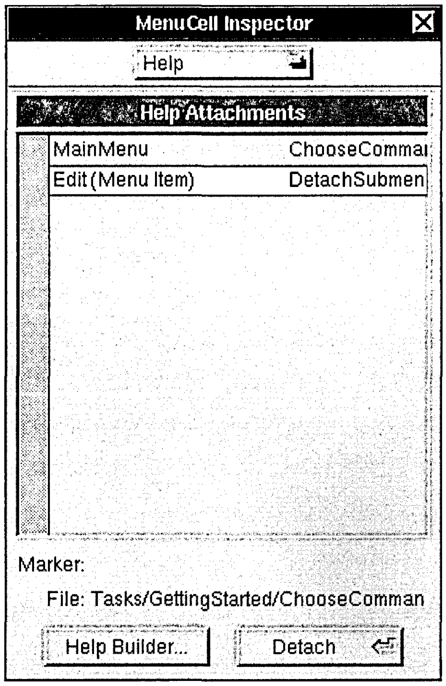
Clicking the horizontal line inside the small square changes the line to a spring shape, indicating that when the superview or window is resized horizontally, the object will also resize to maintain its distance from the left and right margins. In the same way, clicking the vertical line within the object 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. 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.

Reviewing Help Attachments

The Help display lets you review attachments between objects in your application and help text. It also gives you access to Interface Builder's Help Builder panel.



MainMenu ChooseCommai
Edit(Menu Item) DetachSubmen
Marker:
File: Tasks/GettingStarted/ChooseComman

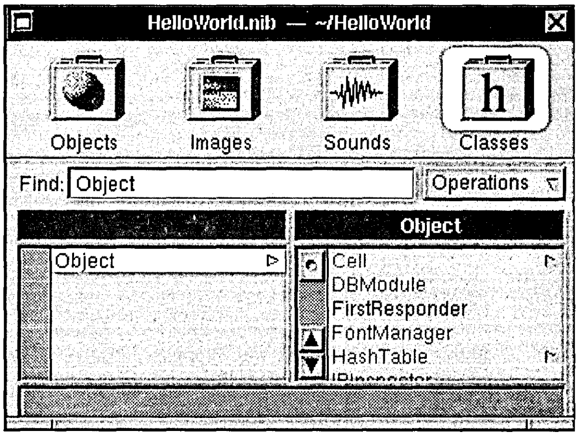
The Help display is used in conjunction with the Help Builder panel. See "Attaching Help to Objects" later in this chapter for information on associating help text with objects in your application.

Assuming you have attached help to objects in your application, the Help display of the Inspector panel will list those attachments. Each entry in the list has two parts. The left half of the entry identifies the object, and the right half displays the file name for the attached help. Below the Help Attachments list are two text fields. The Marker field names the marker that the object is attached to within the help file. If the object isn't attached to any marker in the file, the Marker field is blank. The File field displays the path of the help file relative to the application's **Help** directory. If the entire path isn't visible, scroll the text field horizontally to reveal the hidden portion.

You can remove an attachment by selecting it in the list and clicking the Detach button.

Defining New Classes

The Classes display of the File window shows the classes that are available to your application. It also lets you define new classes. You open this display by clicking the Classes suitcase in the File window.



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. NeXTSTEP classes are displayed in gray, indicating that they can't be edited. The classes you define, being editable, are displayed in black. The Find field helps you locate classes in the hierarchy. Simply enter a class name and press Return. The browser will scroll to the class and select it.

You can add classes to this hierarchy in two ways:

- Drag the icon for a class interface file from the Workspace Manager File Viewer to the File window. When you do, 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.

- Drag to the Subclass button in the pull-down list as described in the next section.

The pull-down list in the Classes display lets you operate on new or existing classes. The commands are described in the following sections.

Subclass

This button creates a new class as the subclass of the class that's currently selected in the browser. New classes are named My*SuperClassName*. The Class Inspector panel opens when you create a subclass allowing you to rename the class and edit its outlets and actions.

Instantiate

This 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 Gauge. If you instantiate additional Gauge objects, they'll be named Gauge1, Gauge2, and so on.

Tip: Although you can instantiate View objects using the Instantiate button, it's generally a better idea to use the CustomView object for this purpose. By dragging a CustomView into your application and reassigning its class, you have an object that can be positioned and sized within a specific window of your application.

Parse

The Parse button displays an Open panel that lets you specify the class interface file you want Interface Builder to parse. Interface Builder reads the specified interface data from the file and then displays the name of the class in its proper location in the browser. Using this command is equivalent to dragging the icon for the interface file into the File window, as described earlier.

Unparse

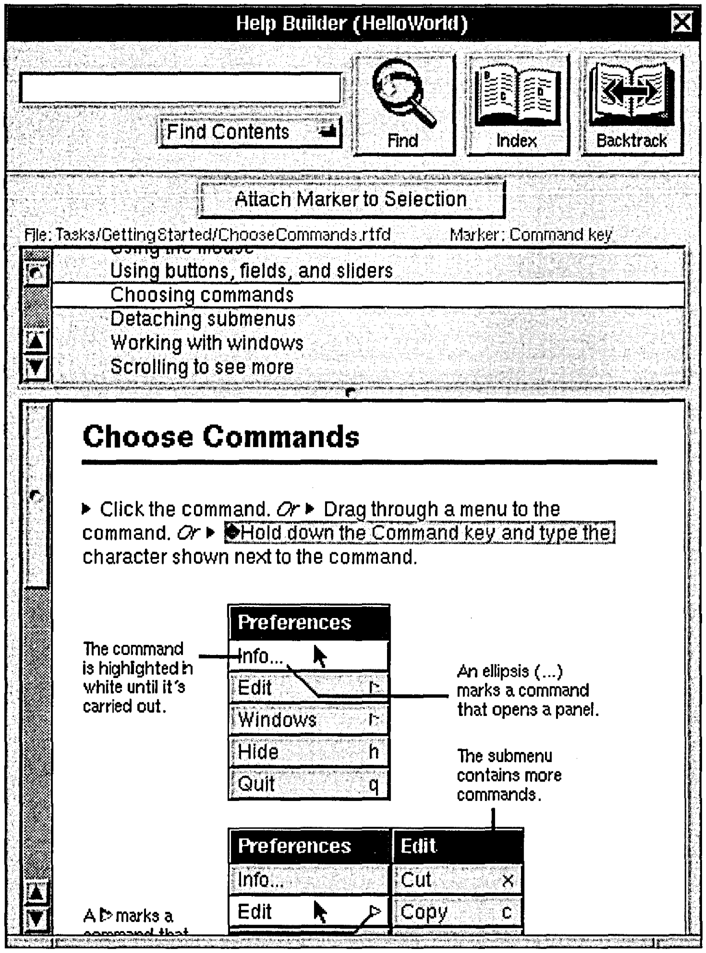
The Unparse button generates template class interface and implementation files for a class you've created with the Subclass command. It writes the interface file based on the outlets and action methods you defined for the class using the Class Inspector. It writes a template implementation file, providing skeletal implementations for each of the class's action methods.

Warning: If you edit the files Interface Builder generates and then reissue the Unparse command, Interface Builder can overwrite the edited files with new template files. Of course, Interface Builder asks for verification before doing so.

Attaching Help to Objects

The Help Builder panel makes it easy to associate help text with any object in your application's user interface. (To learn about the design of the NeXTSTEP help system, see the NXHelpPanel class specification in the *NeXTSTEP General Reference* manual.)

The Help Builder panel is a slightly modified version of the standard Help panel.



File: Tasks/GettingStarted/ChooseCommands.rtfd Marker: Command key

Using buttons, fields, and sliders
Choosing commands
Detaching submenus
Working with windows
Scrolling to see more

Choose Commands

▸ Click the command. *Or* ▸ Drag through a menu to the command. *Or* ▸ Hold down the Command key and type the character shown next to the command.

The command is highlighted in white until it's carried out.

An ellipsis (...) marks a command that opens a panel.

The submenu contains more commands.

A ▷ marks a

Attaching help to an object involves selecting an object in your application, displaying the help text in the Help Builder panel, optionally selecting a help marker within the text, and clicking the Attach... button. Thereafter, when the application runs and the user Help-clicks the object (that is, holds down the Help key and clicks the object), the specified help text will appear in the application's Help panel. However, before you begin attaching help text to your application's objects, you must provide your application with two components: a Help menu item and a **Help** directory.

Interface Builder's Menu palette supplies an Info menu item that, when dragged to your application's main menu, reveals a submenu containing a Help menu item. This menu item is preconfigured to open the Help panel. (If you inspect the Help item's connections, you'll see that it sends a **showHelpPanel:** message to the First Responder object.)

Project Builder can provide your application with the required Help directory. Choose the Add Help Directory command in Project Builder's Project menu to create this directory. Project Builder creates the directory within the ".lproj" directory of your chosen development language (for example, **English.lproj/Help**). It copies into this directory generic table-of-contents and index files.

The next step is to customize these files and to add content files of your own. The generic help text that's accessed through the supplied table-of-contents and index files gives help on basic operations, such as using the mouse and choosing commands. You'll want to add files that describe the operations that are unique to your application. You can also override or eliminate any of the generic help text that isn't applicable to your application.

You create help files using Edit. (Make sure that Edit is in Developer Mode so that the Help commands can be accessed from the Format menu.) Perhaps the easiest way to ensure that the files you add agree in style and formatting with the generic help files is to display a generic file, copy its contents, and paste it into a new Edit document. Be sure to resize the new document's window to the same width as the original so that the text will wrap to the same margins. You can then modify the contents of the new help document and save it in the **Help** directory. If you think you'll want to associate objects with specific passages within the file, rather than to the file in general, you can place help markers within the document.

Each file you add should be represented by a new entry in the table-of-contents file. (However, see the NXHelpPanel class specification for an exception to this rule.) After adding content files, you'll also probably have to update the index.

Once the table-of-contents, content, and index files for your help system are finished, you can begin attaching help to your application's user-interface objects. Display the Help Builder panel by choosing the Help Builder command from Interface Builder's Tools menu or by clicking the Help Builder button in the Help display of the Inspector panel. Select an object in your application's user interface, locate the relevant help text in the Help Builder panel, and click the Attach... button. If the Help inspector is open, it displays this new association in its Help Attachments list.

The Help Builder panel offers several ways to locate specific portions of help text. First, you can use the table-of-contents or index displays to locate a file. In addition, the pop-up list below the Find field lets you search for help files by name, for marker names within the help files, or for any string.

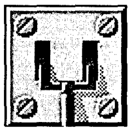
Running Your Application in Test Mode

The Test Interface command puts Interface Builder in test mode. When you choose this command, Interface Builder's supporting windows disappear, leaving only those windows that belong to your application. You can then test the operation of most objects in your application.

Only those objects whose code has been linked into Interface Builder—primarily those objects defined in the Application Kit and those coming from custom palettes—can be exercised in test mode. This means that objects such as Windows, Buttons, and the PrintPanel will operate as they would in a finished application. In test mode, your application's windows can be miniaturized, buttons will highlight when clicked, the Print panel will operate as it should. Objects dragged from custom palettes will also operate normally, since their code is dynamically loaded into Interface Builder when the palette is loaded.

However, objects that have been declared in Interface Builder but whose code hasn't been linked into the Interface Builder application will not work as they would in a finished application. For example, suppose you use Interface Builder's Classes display to create a new class, say the GaugeView class, and then assign the class of a CustomView in your application to be the GaugeView class. When you run the application in test mode, the GaugeView object will not appear since only its Objective C interface—not its code—is available to Interface Builder.

When your application is operating in test mode, Interface Builder's application icon changes to display a large switch.



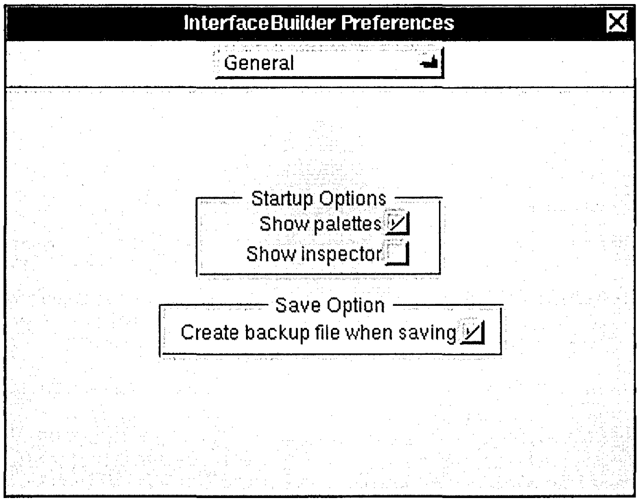
To exit test mode, either choose the Quit command in your application's main menu (if present) or double-click this switch icon.

Setting Preferences

You open the Preferences panel by choosing the Preferences command in the Info menu. This panel has two displays; you use the pop-up list at the top of the panel to access these displays.

General Preferences

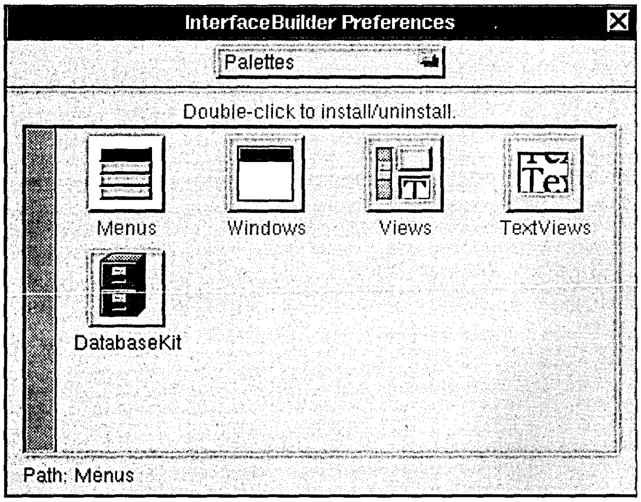
These preferences control which panels appear when Interface Builder is launched and also whether a backup file is created when the nib file is saved.



If the Save Option box is checked, Interface Builder will create a backup file whenever you save a nib file that's been modified. Assuming the box is checked, if you open a nib file named **FindPanel.nib**, make changes, and then save the modified file, Interface Builder will rename the original file **FindPanel.nib~** before saving the modified file as **FindPanel.nib**. Because of the safety of having a backup file, it's generally better to leave this box checked.

Palettes Preferences

This display of the Preferences panel shows you which palettes are available to Interface Builder and lets you control which palettes are installed in the Palettes window.


InterfaceBuilder Preferences
Palettes
Double-click to install/uninstall.
Menus
Windows
Views
TextViews
DatabaseKit
Path: Menus

Each palette is represented by an icon. The palettes that are already installed in the Palettes window display their titles in gray; those that haven't been installed display their titles in black. Double-clicking the icon toggles the state of the palette: If it was installed, it's removed from the panel; if it was uninstalled, it's installed in the panel.

When Interface Builder begins running, it loads the standard palettes (those displayed in the top row of the illustration above) and then loads any palettes it finds in **/NextDeveloper/Palettes**. It also adds to the Preferences display any palettes the user has previously loaded using the Load Palette command. (**Note:** The information about these manually loaded palettes is stored in **~/.NeXT/defaults.nibd**.)

Adding Custom Palettes, Inspectors, and Editors

Interface Builder's primary value as a development tool is that it lets you interact directly with the objects that will make up your application. In general, these objects are defined by the NeXTSTEP system software. However, it's possible to extend Interface Builder's library of objects by creating *custom palettes*, thus letting you interact directly with objects that you or other developers have created.

A custom palette can contain objects of various sorts. Most commonly, a custom palette contains View objects, objects that the user instantiates by dragging into a standard window. It's also possible to create custom palettes that contain MenuCells (which are instantiated by dragging into a menu), Windows (which are instantiated by dragging into the workspace), and other non-View objects (which are instantiated by dragging into the File window).

For any custom palette object, you can provide one or more inspectors. A custom object's inspector appears in the Inspector panel when the user selects the object. Most custom objects will require an Attributes inspector. For example, the fictitious RepeatButton class mentioned earlier would probably require an Attributes inspector to let the user set the repeat rate for a given button. It could also supply its own Connections, Size, and Help inspectors, although the standard versions of these inspectors are generally adequate for most uses.

Finally, a more complex custom object may require its own *editor*. An editor controls how a user can interact with a selected object. Interface Builder itself supplies editors for the objects it knows about. For example, when you double-click a window icon in the File window, Interface Builder's window editor is invoked and brings the actual window to the front. Or, when you double-click a Form object in an application window, Interface Builder's matrix editor is invoked, letting you drag cells to new positions.

An editor that you provide must open its own window when the user double-clicks the custom object. (In this respect, your editor will be like the one provided by the Database Kit for the DBModule object. For a demonstration, load the palette **/NextDeveloper/Palettes/DatabaseKit.palette**, drag a DBModule object into the File window, and double-click the object.) Since each custom object can have its own editor window, editors make copy and paste or drag and drop operations between editor windows possible.

Creating custom palettes, inspectors, and editors involves working with Interface Builder's application programming interface (API). This API is described in detail in Chapter 8 of the *NeXTSTEP General Reference* manual. You should also consult Chapter 18 of this manual for a tutorial describing the process of making a custom palette and inspector.

Interface Builder Command Reference

The remainder of this chapter gives short descriptions of Interface Builder's commands. Only those commands that are unique to Interface Builder are listed; for information on commands that are common to all NeXTSTEP applications see the *User's Guide*.

Commands in the Document Menu

These commands act to open, create, save, or test an Interface Builder document. (Interface Builder documents are generally referred to as "nib files," since that's how they are stored on disk. However, until a document is saved, no file exists, so referring to the document as a "nib file" isn't strictly correct. Even so, for simplicity, Interface Builder documents are referred to as nib files throughout, unless to do so would cause confusion.)

Command	**Description**
Open	Opens an existing nib file.
New Application	Creates a new Interface Builder nib file containing the basic components of an application: a main menu, a standard window, and other resources. You rarely use this command since it's generally more convenient have Project Builder create the nib file for a new application. See the description of Project Builder's New command in the previous chapter for more information.
New Module	Opens the New Module menu, which offers commands for creating various sorts of Interface Builder nib files other than the type used for an application's main nib file. See "Commands in the New Module Menu" later in this chapter for more information.
Save	Saves the current nib file. You can edit more than one Interface Builder nib file at a time. Each open nib file is represented by a File window: The File window that has main or key window status identifies the current nib file.
Save As	Saves the current nib file under a different file name.

Save All	Saves all open nib files.
Revert to Saved	Restores the current nib document to the state represented in the nib file. All changes made since the file was last saved are lost.
Test Interface	Puts Interface Builder in test mode. When you choose this command, Interface Builder's supporting windows disappear, leaving only those windows that belong to your application. You can then test the operation of the objects in your application. See "Running Your Application in Test Mode" earlier in this chapter for more information.

Commands in the New Module Menu

These commands let you create auxiliary nib files of various sorts. (The main nib file is generally created by Project Builder.)

Command	**Description**
New Empty	Creates the simplest sort of nib file, one that includes references only to a File's Owner object and a First Responder object.
New Info Panel	Creates an auxiliary nib file containing a panel that's preconfigured as a standard Info panel.
New Attention Panel	Creates an auxiliary nib file containing a panel that's preconfigured as a standard Attention panel.
New Inspector	Creates an auxiliary nib file containing the components you need when creating an inspector for a custom palette project.
New Palette	Creates an auxiliary nib file containing the components you need for a custom palette project. You rarely issue this command directly, since Project Builder provides this nib for you when you create a new inspector project.

Commands in the Edit Menu

Except for the Set Name command, this menu contains the standard editing commands: Cut, Copy, Paste, Delete, and Select All. These commands work in the expected ways. (See Chapter 15 for a tutorial introduction to Interface Builder's basic commands.)

Command	**Description**
Set Name	Displays a panel that lets you set the name of the selected object. With this name, and the **NXGetNamedObject()** function, you can access objects by name within your application. However, it's generally a better idea to access objects through the use of outlets, since outlets can be connected and disconnected in Interface Builder, eliminating the need to alter your application's code.

Commands in the Format Menu

This menu lets you set the font and formatting attributes of the selected object. It also gives you access to the Layout menu and to the Page Layout panel.

Command	**Description**
Font	Opens the Font menu. Interface Builder's use of the Font menu is entirely standard. By setting the font of a TextField or the Text object within the ScrollView (for example), you are determining which font the user will use in those objects when the application runs.
Text	Opens the Text menu. Interface Builder's use of the Text menu is entirely standard. By setting the text alignment or tab settings of an object in Interface Builder, you are determining the alignment and tab settings for those objects at run time. Note that the ruler commands work only with a Text object that is the document view of a ScrollView.
Layout	Opens the Layout menu, which is described in the next section.
Page Layout	Opens the standard Page Layout panel. This panel lets you specify how the window you print using Interface Builder's Print command will appear on paper. Since a screen pixel is approximately 75 precent 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.

Commands in the Layout Menu

This menu offers commands that help you manage the placement, size, and alignment of View objects that you drag into your application's windows.

Command	**Description**
Bring to Front	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 others. If more than one object is selected when you choose this command, the entire group of objects is brought in front of all other objects in the window.
Send to Back	Puts the selected object or objects behind all other objects in the window.
Size to Fit	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.
Same Size	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	Groups the selected object or objects in 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.
Group in ScrollView	Groups the selected object or objects in a ScrollView. The ScrollView is sized so that it just accommodates the objects in the group. The grouped objects are made subviews of the ScrollView.
Ungroup	Removes the grouping established by the Group or Group in ScrollView commands.

(continued)

Command	Description
Make Row	Aligns the selected objects horizontally. The row extends to the right of the selected object that's nearest the top left corner of the window. The spacing between objects is determined by the original 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	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.
Turn Grid On / Off	Enables and disables the alignment grid in all windows of all open nib files. When the grid is enabled, View objects dragged into a window are constrained in their location and dimensions to the units defined by the grid. By default, the intersections of the grid are aligned, both vertically and horizontally, on every eighth pixel in a window. Also by default, an object's lower left corner is the reference point for alignment with the grid. (The grid spacing and the object's reference point can be changed using the Alignment panel.) 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 / Hide Grid	Displays and hides the alignment grid in all windows of all open nib files. The grid is displayed as a rectangular array of dark gray dots.
Alignment	Opens the Alignment panel, which is described in the "Using the Alignment Panel" section earlier in this chapter.

Resize Window	Makes any application window resizable. During application development, windows that will be resizable at run time display a resize bar. To establish the run-time size of such a window, simply adjust it using the resize bar. Windows that won't be resizable at run time don't display this bar. The Resize Window command lets you adjust the size of such windows by making them temporarily resizable. When you choose this 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 bar disappears.

Commands in the Tools Menu

This menu's commands open or bring to the front the named panel.

Command	**Description**
Colors	Displays the Colors panel.
Inspector	Displays the Inspector panel.
Palettes	Displays the Palettes panel.
Load Palette	Presents an Open panel, enabling you to load additional palettes into Interface Builder's Palette window. See "Setting Preferences" earlier in this chapter for more information.
Help Builder	Displays the Help Builder panel. This panel will be empty unless your application is part of a project containing a **Help** directory. See "Attaching Help to Objects" earlier in this chapter for information on using the Help Builder panel.

4 The Edit Application

4 *The Edit Application*

In addition to the standard UNIX editing tools (**vi**, **ex**, **ed**, and GNU Emacs), the NeXTSTEP development environment provides a mouse-based text editor named Edit for creating and editing ASCII or RTF (Rich Text Format®) text files.

Edit has all the standard features of a text editor: You can type paragraphs of text without pressing the Return key (the text wraps automatically at the end of each line, and if you change fonts or resize the window, the text rewraps accordingly). You can use the mouse to select where text will be entered and to select text you want to edit. And you can find and replace text, move and copy it, and so on.

While Edit has the functionality of a good text editor, it's particularly suited for writing programming code and performing other application-development tasks. It lacks many of the capabilities found in similar applications, but it has many features specifically designed for programmers. For example, Edit supports name expansion, folder browsing, block nesting in program listings, and a structured editing facility. It also provides interapplication functionality with Project Builder, Terminal, and the GDB debugger.

Starting Edit

You can start Edit from the workspace as you would start up any other application. Alternatively, you can start up Edit from a shell window by typing the following command at the UNIX prompt:

Edit [*file name* ...] **&**

Several command-line options allow you to override various default characteristics of Edit for the work session you're about to start—characteristics such as the number of lines and columns in new windows, the font family used, and the font size. For example:

```
Edit -NXFont Times-Roman Fruit.m &
```

These command-line options can be specified in any order, as long as they precede any file names. Several options are listed below.

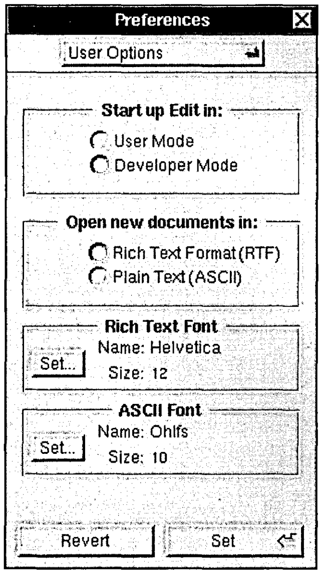
Option	**Effect**
IndentWidth	Specifies the width of indentation for block nesting. The default value is 4.
NXFont	Specifies the font family. The default font is Helvetica®.
NXFontSize	Specifies the font size, in points. The default value is 12.
Tags	Specifies one or more pathnames to **tags** files that will be searched by the Source command. The pathnames should be separated by a colon, as in a standard UNIX path list. The default is "tags," which indicates that the **tags** file in the current folder will be searched. See the description of using **tags** files under "Interacting with UNIX" later in this chapter for more information about using **tags** files in Edit.
DeleteBackup	Specifies whether the previous version of a file is deleted or retained as a backup when you save changes to the file. The default value is YES, which means that the previous version is deleted. If the previous version is saved as a backup, its name is the same as the original file name, but with a tilde (~) appended to the name.
NXMenuX	Specifies the (positive) distance in pixels from the left edge of the screen to the left edge of the main menu.
NXMenuY	Specifies the (positive) distance in pixels from the bottom of the screen to the top of the main menu.

Edit will use the default value for each option unless you override it with a command-line option. The value specified in the command line will remain in effect only for the work session you're about to start. The next time you use Edit, the defaults will go back into effect.

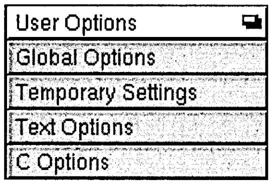
You can set new default values for each of the above characteristics (except for screen coordinates) using the Preferences panel, which is described in the following section. Most defaults set with the Preferences panel remain in effect until you change them.

Setting Preferences

The Preferences command in the Info menu displays the Preferences panel, shown below. The Preferences panel lets you set default values for various Edit options. For example, you can set default font properties or specify the size of new windows.


Start up Edit in:
User Mode
Developer Mode
Open new documents in:
Rich Text Format (RTF)
Plain Text (ASCII)
Rich Text Font
Name: Helvetica
Size: 12
ASCII Font
Name: Ohlfs
Size: 10

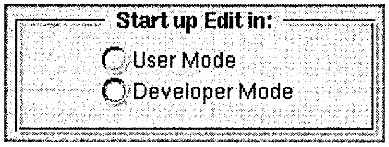
Enter values and click buttons to specify new preferences, as described below. Then click Set to set the new preferences (or click Revert to restore the previous settings). In general, the new settings remain in effect until you change them. However, you can temporarily override some of the defaults by starting up Edit from a shell window and specifying one or more command-line options (as described earlier under "Starting Edit").



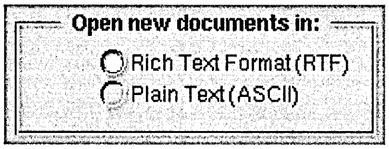
You can press the button labeled User Options and, in the list that appears, choose from several other sets of options that are available. These other options are described below, after the user options.

User Options

Choose User Options in the Preferences panel's pop-up list to see the user options that can be specified. User options are saved in your defaults database and continue to be used until you specify different values for them.

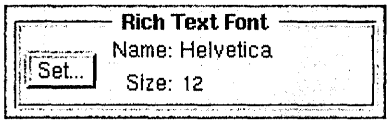


By default Edit starts up in User mode, which presents just a subset of the commands available in Developer mode. If you're using Edit for application development be sure to click the Developer Mode button.



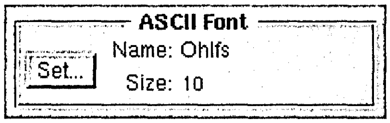
Click one of these buttons to specify whether new documents are created in RTF (Rich Text Format) or ASCII format.

Tip: After you've created or opened a document, you can change its format by choosing the Make Rich Text command or the Make ASCII command in the Text menu.



Name: Helvetica
Size: 12

Click the Set Button in the Rich Text Font field to set a default font for Edit windows that are in RTF format. Specify the font family, typeface, and size in the Font panel that appears, and click the Set button in the Font panel when you're done. After you save these settings, all subsequently created RTF documents will by default display text in the specified font.



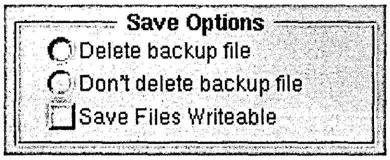
Name: Ohlfs
Size: 10

Click the Set Button in the ASCII Font field to set a default font for Edit windows that are in ASCII format. Specify the font family, typeface, and size in the Font panel that appears, and click the Set button in the Font panel when you're done. After you save these settings, all subsequently opened Edit windows that containing ASCII files will display text in the specified font.

Tip: When working with code or UNIX command output, it's best to use a fixed-width font family, such as Courier.

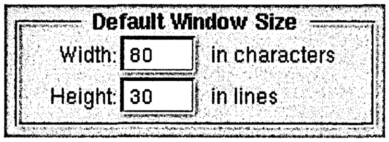
Global Options

Choose Global Options in the Preferences panel's pop-up list to see the global options that can be specified. Global options are saved in your defaults database and continue to be used until you specify different values for them.

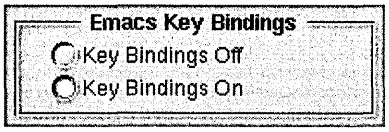


When the "Delete backup file" option is selected, Edit automatically deletes the previous version of a file when the current version is saved. Click "Don't delete backup file" to retain the previous version of a file when you save the current version (if the previous version of a file is saved). This backup file is saved under the original file name, but with a tilde (~) appended to the name.

If you try to save a file that's write-protected, you can do so by responding affirmatively to the confirmation panel that appears (as long as you own the file). Check the Save Files Writeable button if you want such write-protected files to lose their write-protected status when they're saved.


Default Window Size
Width: 80 in characters
Height: 30 in lines

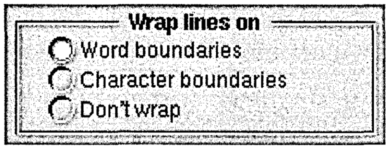
To set a default size for Edit file windows, enter a width (in number of characters) in the Width field and a height (in number of lines) in the Height field. Edit files that you open after saving these settings will be displayed in windows with the dimensions you specify. (Note that since these dimensions are specified in characters and lines, the default window size will be affected by the default font.)


Emacs Key Bindings
Key Bindings Off
Key Bindings On

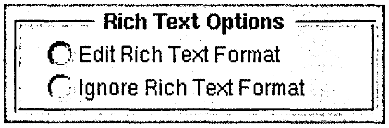
Click one of the buttons in the Emacs Key Bindings field to specify whether or not Emacs key bindings are enabled. For a list of the Emacs key bindings available in Edit, see "Using Keyboard Editing Commands."

Temporary Settings

Choose Temporary Settings in the Preferences panel's pop-up list to see the temporary settings that can be specified. These are called temporary settings because they're not saved in your defaults database.


Wrap lines on
Word boundaries
Character boundaries
Don't wrap

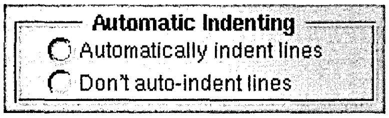
When the "Word boundaries" option is selected, text wraps onto the following line at the end of each full line, but no words are split across lines. Clicking "Character boundaries" also causes text to be wrapped at the end of each line, but words can be split across lines. Clicking "Don't wrap" causes text to not wrap at all.


Rich Text Options
Edit Rich Text Format
Ignore Rich Text Format

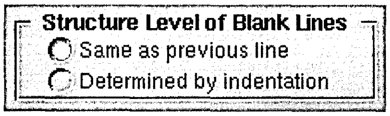
When the Edit Rich Text Format option is selected, RTF files that you open are displayed as formatted text. Click Ignore Rich Text Format to view RTF files as unformatted text with the format commands visible. Because other applications use Edit to view formatted text, you should normally leave the Edit Rich Text Format option selected.

Text Options

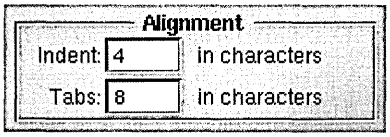
Choose Text Options in the Preferences panel's pop-up list to see the text options that can be specified. Text options are saved in your defaults database and continue to be used until you specify different values for them.


Automatic Indenting
Automatically indent lines
Don't auto-indent lines

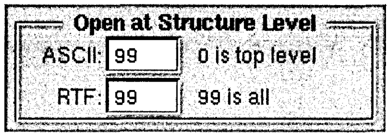
When the "Automatically indent lines" option is selected, Edit indents each new line the same as the line above it (automatic indentation is useful for typing indented lines of code). Click "Don't auto-indent lines" if you want each new line to start at the left margin.


Structure Level of Blank Lines
Same as previous line
Determined by indentation

When the "Same as previous line" option is selected, Edit assigns each "blank" line (that is, each line that contains no visible text) the same structure level as the previous line. Click "Determined by indentation" if you want the structure level of blank lines to be determined by the amount of indentation (that is, tabs and spaces) on that line, rather than by the indentation of the previous line.


Alignment
Indent: 4 in characters
Tabs: 8 in characters

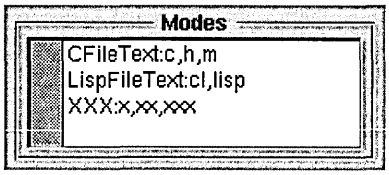
In the Indent field, enter the number of characters you want to shift right or left with the Text menu's Nest and Unnest commands. In the Tabs field, enter the number of characters you want between tab stops.



ASCII: 99 0 is top level

RTF: 99 99 is all

In the ASCII and RTF fields, enter a number between 0 and 99 to specify how many levels of structure will be visible in a newly opened file of that type. A 0 indicates that only the top level of text (that is, text that's flush left) will be visible, a 1 indicates that the first sublevel of text should also be visible, and so on.

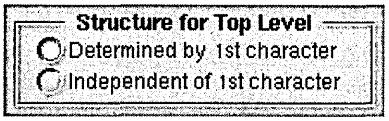


```
CFileText:c,h,m
LispFileText:cl,lisp
XXX:x,xx,xxx
```

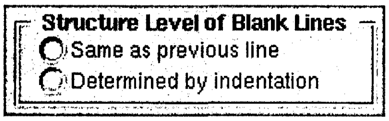
In addition to the default Text mode, there are two editing modes for C and Lisp source files (these modes optimize some minor aspects of Edit's behavior for use with each of these programming languages). You can specify in the Modes field any additional file extensions that you want associated with either of these two modes.

C Options

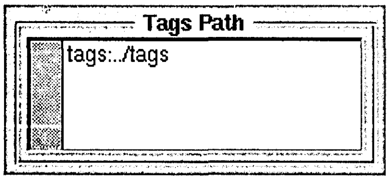
Choose C Options in the Preferences panel's pop-up list to see the C source code options that can be specified. C options are saved in your defaults database and continue to be used until you specify different values for them.



When the "Independent of 1st character" option is selected, commands in the Structure menu operate solely on the basis of indentation, independent of particular characters. Click "Determined by 1st character" if you want Structure menu commands to treat C preprocessor directives (lines whose first character is #) specially—that is, as second-level text, rather than top-level.


Structure Level of Blank Lines
○ Same as previous line
○ Determined by indentation

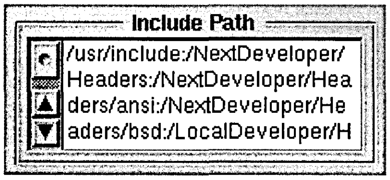
When the "Same as previous line" option is selected, Edit assigns each "blank" line (that is, each line that contains no visible text) the same structure level as the previous line. Click "Determined by indentation" if you want the structure level of blank lines to be determined by the amount of indentation (that is, tabs and spaces) on that line, rather than by the indentation of the previous line.


Tags Path
tags:../tags

In the Path field, enter the pathname of one or more **tags** files that you want Edit to search when you choose the Source command in the Utilities menu. A **tags** file, which you create using the UNIX **ctags** command, contains the locations of program object definitions among a given group of files. The Source command searches the **tags** files specified here for the location of an object definition and then opens the file containing the definition.

If you leave the default entry of "tags:../tags" in this field, Edit will search only the **tags** files in the current folder (the folder containing the file in the main window) and in the current folder's parent folder. You can replace or add to the default, however, by entering the pathnames of one or more other **tags** files; you separate multiple pathnames with a colon as in a standard UNIX path list.

See the description of the Source command in "Commands in the Utilities Menu" later in this chapter for more information about using Edit's Source command with **tags** files.


Include Path
/usr/include:/NextDeveloper/Headers:/NextDeveloper/Headers/ansi:/NextDeveloper/Headers/bsd:/LocalDeveloper/H

The Include Path field displays your default include path (the path the preprocessor uses to search for system header files). You can redefine this path by editing the text and then clicking the Set button.

Performing Basic Operations

This section summarizes several basic Edit concepts and operations. For more information about basic operations common to Edit and other standard NeXTSTEP applications, see the *User's Guide*.

Opening Edit Files

In addition to opening Edit files from the workspace, you can open them from within Edit by using the Open or Open Selection commands in the File menu. (These commands are described later in the chapter.)

An alternate way to open one or more files is to use Edit's **openfile** command at the UNIX prompt in a shell window. You can specify one or more file names (or pathnames), which are interpreted relative to the shell window's current folder. For example, the following command would open all the files in the current folder that end with a ".c" extension, plus all the files in a subfolder called headers that end with a ".h" extension:

```
openfile *.c headers/*.h
```

Each file is opened in its own Edit window. Note that the **openfile** command can be used only when Edit is running.

Using File Windows and Folder Windows

Edit provides two types of standard windows: *file windows* and *folder windows*. As in other applications, there are also panels and menus.

Note: Unless otherwise specified, folder windows mentioned in this chapter are Edit folder windows, not Workspace Manager folder windows.

An Edit file window displays a document file that you can view and edit. When you make changes to text displayed in a file window, the version of the file on the disk isn't affected until you save the file with the File menu's Save command. When a file contains unsaved changes, the window's title bar displays a partially drawn close button. If you miniaturize a window containing unsaved changes, its miniwindow is highlighted in gray.

An Edit folder window displays a list of the files and subdirectories contained in a folder. You don't edit the contents of a folder window; instead, you use the displayed folder listing to find and select other files or directories to open.

Two special features are available in Edit folder windows:

- You can type a character to find and select the first item starting with that character. Each additional character you type deselects the previously selected item and finds the first item starting with the newly typed character. The commands in the Find menu can also be used to find and select items in a folder window.
- You can double-click a file or folder name to open an Edit window displaying that file or folder. This is equivalent to selecting the name and choosing the Open Selection command in the File menu.

You can also open an Edit folder window by choosing the Open Folder command in the File menu. The command displays a panel in which you enter the pathname of a folder to be opened.

Selecting Text

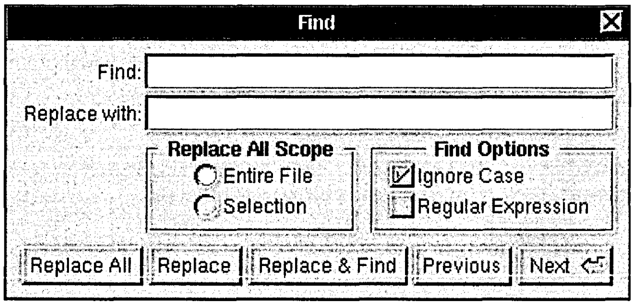
Most operations in Edit are performed on the current selection, which appears either as the insertion point (a blinking vertical bar) or as highlighted text.

You make selections using the standard selecting techniques: Position the insertion point by clicking, and select a block of text either with multiple-clicks or by dragging with the mouse, as outlined below.

Method	**Effect**
Clicking	Positions the insertion point where you click.
Dragging	Selects text that you drag across. To select beyond what's currently displayed, drag past the edge of the window. The contents scroll automatically and text continues to be selected.
Shift-clicking	Selects from the insertion point, or extends or shortens a selection.
Double-clicking	Selects a word. If you double-click one of a pair of matching delimiters (parentheses, braces, or square brackets) the pair of delimiters and the enclosed text are selected.
Triple-clicking	Selects a line or paragraph.

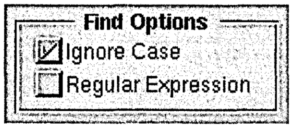
Finding and Replacing Text

The Find Panel command opens a panel that lets you locate the next occurrence of a specified text string and optionally replace it with another string.



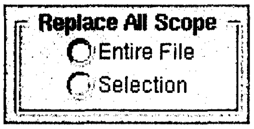
In the Find field, specify the text to be located. You can't type tab or return characters in the Find field, because of their other functions: Pressing tab moves the insertion point to the "Replace with" field, and pressing Return begins the search for the text. To specify a tab character in the text, type Alternate-Tab. Likewise, type Alternate-Return to specify a carriage return character.

In the "Replace with" field, you may specify a replacement string. Then click one of the panel's buttons to perform the exact search operation you want, as described below. If the end of the document is reached during a search, Edit continues searching from the beginning of the document. When searching backward and reaching the beginning of the document, Edit continues searching from the end.

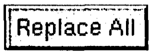


When Ignore Case is checked, Edit doesn't distinguish between uppercase and lowercase letters when finding a match during the search. If Ignore Case is not checked, the search is case-sensitive.

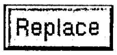
If the Regular Expression box is checked, Edit interprets the text in the Find field as a UNIX regular expression (see the UNIX manual page for **ed** for information on regular expressions). If this box is unchecked, the Find entry is taken as a literal string of text.



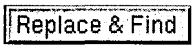
The Replace All Scope options specify whether Replace All applies to the entire document (Entire File) or only to the current text selection (Selection).



In the area that you specify, the Replace All button replaces all occurrences of the text entered in the Find field with the text entered in the "Replace with" field. If the "Replace with" field is blank, Replace All deletes all occurrences of the text. After a search with Replace All, the Find panel reports the number of occurrences that were replaced.

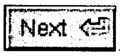


After text has been found, click Replace if you want to replace the current selection with the text in the "Replace with" field (or if the "Replace with" field is blank and you want to delete the current selection).



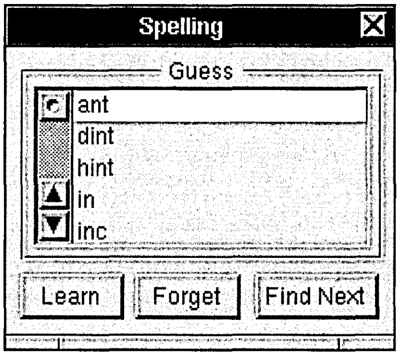
Click this button to replace the current selection and find the next match. This button is a shortcut to using the Replace button and then the Next button.

Click the Previous button to find the first occurrence of the Find entry searching backward from the insertion point or the beginning of the current text selection.



Click the Next button to find the first occurrence of the Find entry searching forward from the insertion point or from the end of the current selection. (Pressing the Return key has the same effect, but with one difference: If you used the Find Panel command's keyboard alternative to display the panel, pressing the Return key causes the panel to disappear instead of remaining on the screen.)

Checking Spelling

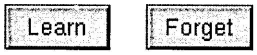


The Spelling command opens a panel that lets you check the spelling of words, choose from possible corrections, and modify the spelling dictionary. As a convenience, Edit doesn't open the Spelling panel as the key window, so that you can type to correct a misspelling without having to click in the file window first.

To begin a spelling check from this panel, click Find Next. Spelling locates and selects the next word not contained in the spelling dictionary. (Edit uses a systemwide 100,000-word spelling dictionary that's shared by other applications, such as Mail.)

The search for misspelled words is circular, so that all the text in the main window is searched. The search starts at the word containing the insertion point, or at the last word in the current selection, and goes to the end of the text. If no potentially misspelled words are found, the search continues at the beginning of the text until it comes back to the starting point.

The Spelling panel displays a list of possible corrections to the last word selected as misspelled (unless the word is completely unrecognizable). Double-clicking one of them will replace the selected word in the main window with the desired correction.

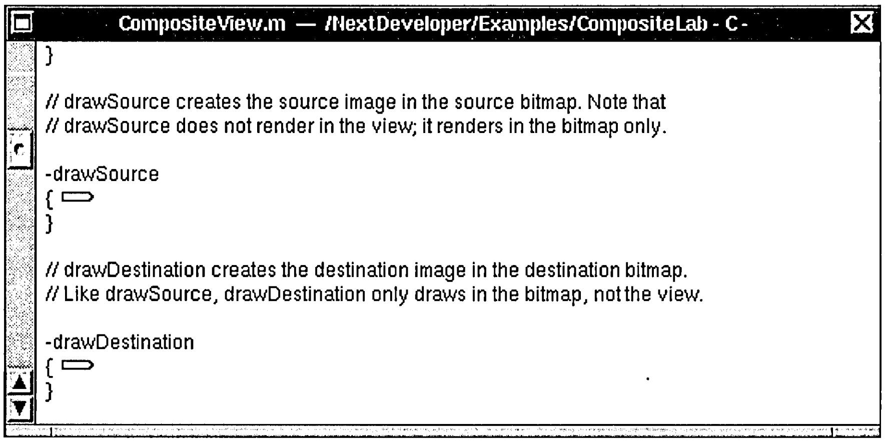


The Learn and Forget buttons let you remove or add words from the spelling dictionary. If a correctly spelled word is identified as misspelled, you can add it to the dictionary by clicking Learn. You can also remove any word you've added to the dictionary, by selecting it and clicking Forget.

To search for the next misspelled word, click Find Next (or choose the Check Spelling command from the menu).

Contracting and Expanding Text in a File Window

Edit provides a Structure capability that lets you quickly move around in C files (as well as in any other type of file where levels of structure are represented by varying degrees of indentation—outlines, for example). Commands in the Structure menu can be used to "contract" text in the main window, displaying only the text at a particular level of indentation. Text that's indented beyond that level is hidden. Figure 4-1 shows a document that's been contracted—only the top-level lines (those that are flush left) are visible. Notice the two white text arrows, which indicate the presence of contracted text.


CompositeView.m — /NextDeveloper/Examples/CompositeLab - C -

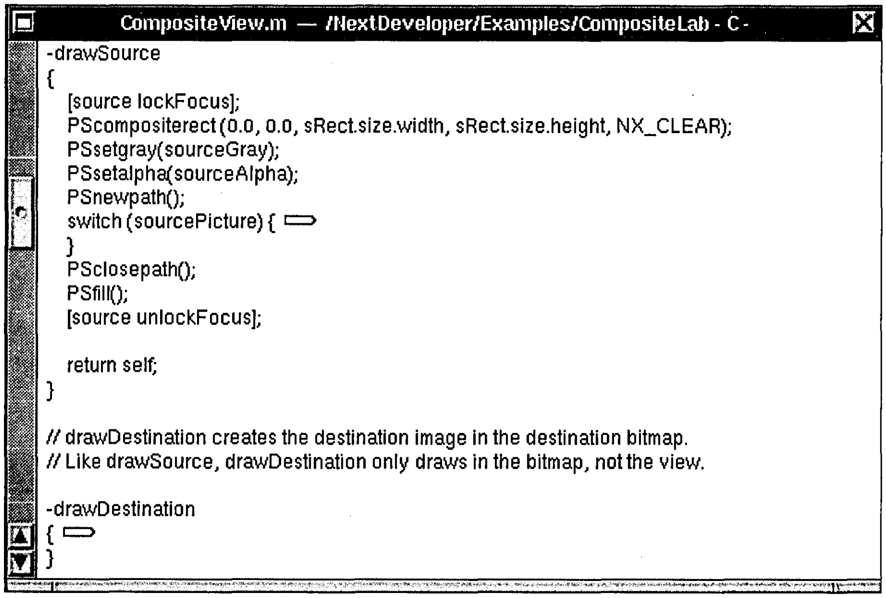
```
}

// drawSource creates the source image in the source bitmap. Note that
// drawSource does not render in the view; it renders in the bitmap only.

-drawSource
{
}

// drawDestination creates the destination image in the destination bitmap.
// Like drawSource, drawDestination only draws in the bitmap, not the view.

-drawDestination
{
}
```

Figure 4-1. File Window with Just First-Level Text Expanded

When text is contracted, only the display is changed—the document itself (including font changes and text properties) remains unchanged. However, while some Edit commands affect both the expanded and the contracted portions of the document (for example, Cut and Paste), other commands affect only the portions of the document that are expanded (for example, commands that change the font).

Commands in the Structure menu let you expand or contract either the entire contents of the window, or just the current selection. The rest of this section describes some mouse shortcuts that you'll probably use even more frequently than the menu commands.

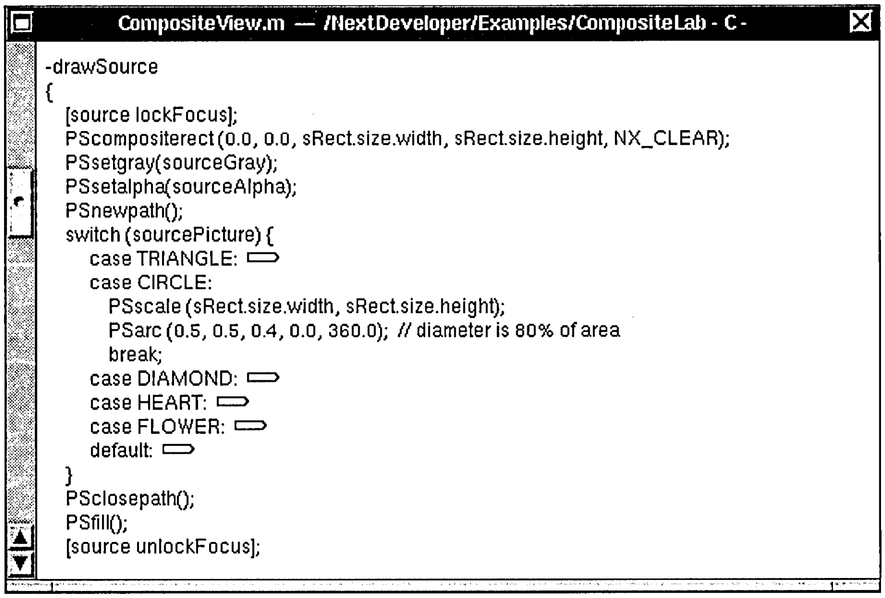
Clicking a text arrow expands (that is, displays) the text that the arrow represents. Control-clicking a text arrow expands just the top level of the text that the arrow represents. For example, Figure 4-2 shows what the **drawSource** definition looks like after Control-clicking the first of the two text arrows shown in Figure 4-1. Notice that the **drawSource** definition has expanded, but the **drawDestination** definition is still contracted. Also notice that the **drawSource** definition hasn't expanded completely—the **switch** statement contains yet another level of contracted text.


CompositeView.m — /NextDeveloper/Examples/CompositeLab - C -

```
-drawSource
{
  [source lockFocus];
  PScompositerect(0.0, 0.0, sRect.size.width, sRect.size.height, NX_CLEAR);
  PSsetgray(sourceGray);
  PSsetalpha(sourceAlpha);
  PSnewpath();
  switch (sourcePicture) { ⇨
  }
  PSclosepath();
  PSfill();
  [source unlockFocus];

  return self;
}

// drawDestination creates the destination image in the destination bitmap.
// Like drawSource, drawDestination only draws in the bitmap, not the view.

-drawDestination
{ ⇨
}
```

Figure 4-2. File Window with Some Second-Level Text Expanded

Figure 4-3 shows the **drawSource** definition after Control-clicking the **switch** statement's text arrow. Each **case** statement in the **switch** contains an additional level of contracted text. The text for CIRCLE, however, isn't contracted—it's already been expanded by clicking (or Control-clicking) its text arrow.


CompositeView.m — /NextDeveloper/Examples/CompositeLab - C -

```
-drawSource
{
  [source lockFocus];
  PScompositerect(0.0, 0.0, sRect.size.width, sRect.size.height, NX_CLEAR);
  PSsetgray(sourceGray);
  PSsetalpha(sourceAlpha);
  PSnewpath();
  switch (sourcePicture) {
    case TRIANGLE: ⇨
    case CIRCLE:
      PSscale (sRect.size.width, sRect.size.height);
      PSarc (0.5, 0.5, 0.4, 0.0, 360.0); // diameter is 80% of area
      break;
    case DIAMOND: ⇨
    case HEART: ⇨
    case FLOWER: ⇨
    default: ⇨
  }
  PSclosepath();
  PSfill();
  [source unlockFocus];
```

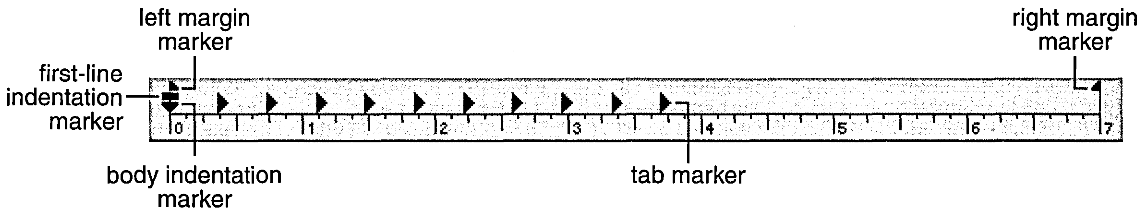
Figure 4-3. File Window with Some Third-Level Text Expanded

If you want to recursively expand all the sublevels of text represented by a text arrow, click the arrow instead of Control-clicking it.

Control-clicking anywhere within an indented block of text contracts the text.

Using the Ruler

Edit provides a *ruler* that can be used to alter the format (margins, indentation, and tab stops) of text in a file window. The Text menu (a submenu of the Format menu) contains commands for showing the ruler and copying ruler settings, as well as commands for centering or otherwise aligning text between the margins.



To display the ruler, choose the Show Ruler command from the Text menu (this command is enabled only if the file window contains text in RTF format). The ruler settings show the format of the paragraph that contains the insertion point or the beginning of the selected text.

You can move margin, indentation, and tab markers by dragging them along the scale of the ruler. When you move a marker in the ruler, a vertical gray line appears, running from the marker to the bottom of the window. This line makes it easier for you to determine the position of the marker relative to the text.

There are two important things to note about the margin settings:

- The left and right margin settings affect the entire text. Thus the margin settings, whatever they may be, will always be uniform throughout a file.
- The right margin adjusts to match the width of the window: If you resize the window wider, the right margin marker moves to the right and the lines of text become longer; narrowing the window moves the right margin marker to the left.

Tab stops and indentation may be customized for individual paragraphs. Unless you specifically change the tab stops and indentation, each new paragraph you type will have the same tab stops and indentation as the preceding one. If you move or copy a paragraph (including the Return at the end of it), the paragraph will keep its original tab stops and indentation.

If you want to change the tab stops or indentation of a single paragraph, you need only click in the paragraph; you don't have to select the entire paragraph. After you make your changes, the paragraph becomes selected. When you're ready to type again, just position the insertion point where you want to enter text.

When several paragraphs are selected, the ruler displays the format of the first one. If you then change a ruler setting, the selected paragraphs will receive not only that ruler setting, but all the formatting of the first paragraph. You can also copy the format of one paragraph to other paragraphs with the Copy Ruler and Paste Ruler commands in the Text menu.

Note: If you copy formatted text from Edit into another application, the formatting will be copied along with the text only if the application can interpret RTF.

Margins ↱ ↰

The margin markers determine the left and right margins of the entire Edit file. To set the left or right margin, drag the corresponding margin marker to the desired position on the ruler. As you drag the left margin marker, the tab and indentation markers move with it, remaining the same distance relative to the left margin.

Indentation ⊤ ▾

There are two indentation markers:

⊤ The first-line indentation marker indents the first line of a paragraph.
▾ The body indentation marker indents all the rest of the lines of the paragraph.

The two indentation markers move independently; adjusting one does not affect the other. Initially, both indentation markers are aligned with the left margin marker. Neither indentation marker can be moved to the left of the left margin marker.

The relative positions of the two indentation markers determine the style of paragraph indentation:

- Dragging the first-line indentation marker to the right of the body indentation marker creates a regular paragraph indentation.

- Dragging the first-line indentation marker to the left of the body indentation marker creates a *hanging indent.*

- Dragging both the first-line and the body indentation markers to the same position indents the entire paragraph.

Changing the left margin of the text doesn't affect indentation. Both indentation markers move with the left margin marker, maintaining the same distance from it.

Tabs ▸

Tab markers set the locations of tab stops—the positions that the insertion point will advance to if you press the Tab key. Typing proceeds normally (from left to right) after the tab, which lets you align columns of text vertically along the left side.

Initially, the ruler displays ten tab markers set eight spaces apart. Note that these initial tab markers may not line up exactly with the calibration marks on the ruler's scale.

To reposition a tab stop, drag the tab marker to the desired position on the ruler. To create a new tab marker, click below the scale of the ruler: The marker will appear on the ruler above where you clicked. You can remove a tab marker by dragging it off the left or right end of the ruler.

Like indentation, tab stops adjust accordingly when you move the left margin marker.

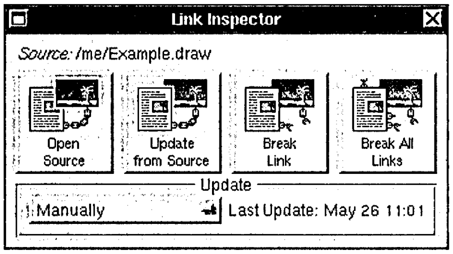
Adding Linked Graphics

You can add linked graphic images to an Edit document, so that whenever the original images are modified, the linked copies you added can be updated automatically.

To link graphics from another application into Edit, the application used to create the graphic image must be able to supply linked information. The Draw application (in **/NextDeveloper/Demos**) is an example of such an application. (Some applications can supply linked information that isn't a graphic image, such as text or database information. You add this information to a document in the same way that you add linked graphics.)

To paste a linked graphic in an Edit document, copy the graphic in the source application and then choose the Paste and Link command in Edit's Link menu.

The Link Inspector command (also on the Link menu) opens the following panel, which you use to maintain and update the links you create. Using this panel you can open the source document, update the linked graphic, break the selected link, or break all links. In addition, you can specify how to update the link when changes to the source graphic occur.



Source: /me/Example.draw

Manually

Last Update: May 26 11:01

For more information about working with linked graphics, see Chapter 11, "Working with Graphics," in the *User's Guide*.

Adding Help Links

The Help menu in Edit provides commands that are used to add or edit Help links. Although Help links are designed for use within an application's on-line Help system, they can also be used more generally (for example, the Contents file for the on-line developer release notes contains links to the various release note files). For information about adding a Help system to an application you're developing, see Chapter 3.

To work with Help links and markers, use the following commands in the Help menu (choose the Help command in the Format menu):

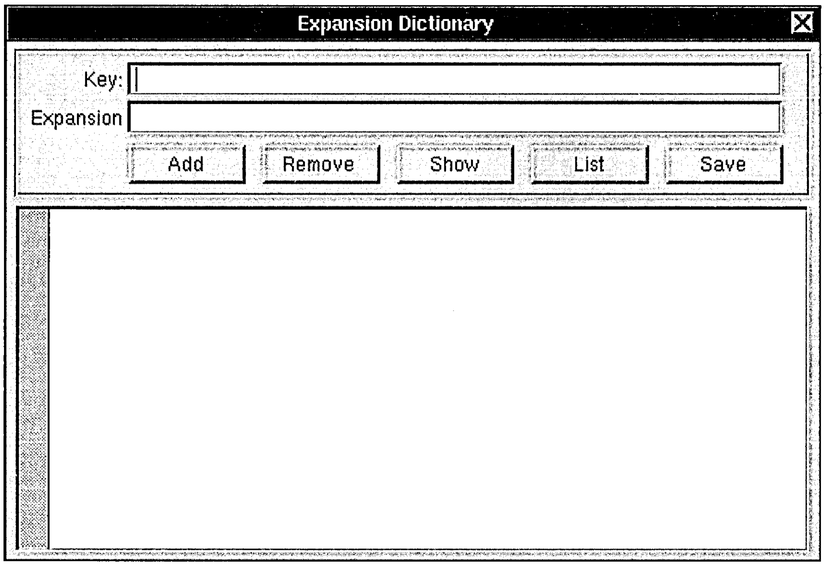
- Choose Insert Link to insert a Help link at the current insertion point. In the Link Inspector that appears, specify the name of a file and (optionally) a marker in that file.
- Choose Insert Marker to insert a Help marker at the insertion point in the main window. A Marker panel appears in which you specify a name to associate with the marker. When you insert a link to the marker, you'll identify it by this name.
- Choose Show Markers to show all the markers in the main window, or Hide Markers to hide them.

If you want to edit a link or marker you've created, Command-click it to bring up the Inspector panel. To delete a link or marker, select it and press the Delete key, just as you would with text.

Using Templates

Three commands on the Expert menu—Expansion Dictionary, Insert Field, and Next Field—let you for create and use glossary entries. Glossary entries are abbreviations for commonly used text strings or templates that you can type and then expand into the full text entry with a single keystroke.

To define a glossary entry, open the Expansion Dictionary panel by choosing Expansion Dictionary in the Expert menu.



In the Key field, enter an abbreviation for the text string or template. In the Expansion field, enter the expanded text that you want the abbreviation to represent. If you want the expansion to occupy more than one line, press Alternate-Return while typing in the Expansion field to insert Return characters between lines. Note that when you press Alternate-Return, the line of expanded text you just typed disappears from the field, leaving room to type the next line.

To use a glossary entry, type the abbreviation in a document and then press the Escape key; the abbreviation is replaced by its expansion. For example, if you frequently need to type **setOutputForm**, you could use the Expansion Dictionary command to associate the abbreviation "sof" with the longer declaration. To enter **setOutputForm**, you would only have to type **sof** and press Escape. The abbreviation doesn't even have to be typed in full for the expansion to occur, as long as what you do type refers unambiguously to a glossary entry.

If you're using the Expansion Dictionary window to create a template containing fields you'll be editing after the text is expanded, surround each field with European quotes («»), as described below. For example:

```
Subject:  «subject»
To:  «recipient»
cc:  «cc»»

«message»
```

You can enter European quotes in the Expansion field by choosing the Insert Field command, or you can enter them directly from the keyboard by typing Alternate-(and Alternate-). After inserting the template into a document, you can quickly find each editable field by choosing the Next Field command, which positions the insertion point at the next field in the template.

Add

After entering the abbreviation and the expanded text it stands for in the Key and Expansion fields, click the Add button to accept the new glossary entry.

Save

Then to actually save the entry (so that it's there for the next work session), click Save.

Remove

To remove a glossary entry, type its abbreviation in the Key field and click the Remove button.

Show

You can view the expanded text associated with an abbreviation by entering the abbreviation in the Key field and then clicking Show.

List

Click List to view a list of all available glossary entries.

Using Keyboard Editing Commands

In addition to letting you edit text using menu commands (and their keyboard equivalents), Edit also supports several Emacs-style editing commands that can be typed from the keyboard. The table below lists the key combination corresponding to each of these commands and a description of what the command does.

Command	**Action**
Control-B	Moves back one character
Control-F	Moves forward one character
Alternate-b	Moves back one word
Alternate-f	Moves forward one word
Control-A	Moves to beginning of line
Control-E	Moves to end of line
Control-D	Deletes next character
Control-H	Deletes previous character
Alternate-d	Deletes to end of current (or next) word
Alternate-h	Deletes to beginning of current (or previous) word
Control-K	Deletes forward to end of line
Alternate-<	Moves to beginning of text
Alternate->	Moves to end of text
Control-N	Moves down one line
Control-P	Moves up one line

Interacting with UNIX

Edit provides some useful commands for using UNIX utilities from within Edit. These include:

- Two commands for piping output from UNIX commands directly into Edit files
- A Source command that you can use with one or more **tags** files to locate program objects in a group of files

Piping UNIX Output to a File

Edit lets you pipe the output of a UNIX command directly into an Edit window. This is a useful technique for inserting output from other applications into your own programs.

For example, to produce a 1992 calendar in an empty window, choose Command in the Utilities menu, enter

```
cal 1992
```

in the panel that appears, and press Return. The output appears in an untitled window.

If instead you wanted the calendar to appear in the main window, position the insertion point where you want the calendar to appear (or select what you want it to replace). Then choose Pipe in the Utilities menu. Enter the same command as before and press Return. This time the output appears in the main window at the insertion point or in place of the current selection.

You can also use the Pipe command to manipulate the current text selection with another UNIX program. If the command accepts input, the selection will be used as input—for example, you could sort the selection with the **sort** command.

If there are Command and Pipe commands that you use frequently, you can define them as menu items in the User Commands and User Pipes submenus in the Utilities menu. To do this, enter a definition for each command in a file named **.commanddict** or **.pipedict** in your home folder.

Each command definition contains at least two fields, separated by tabs:

command name<tab>*command definition*

For example, the following entry defines a Pipe command called Sort Selection, which runs the UNIX sort command using the current selection as input:

```
Sort Selection      sort -
```

One additional field (inserted between the two required fields and separated from them by tabs) can be used to specify a keyboard alternative for the command. For example, this definition of the Sort Selection command assigns to it the keyboard alternative Command-5:

```
Sort Selection      5          sort -
```

If you make changes to your **.commanddict** or **.pipedict** file while Edit is running, you must quit and restart Edit in order for your changes to appear in the User Commands or User Pipes menu.

Two special variables can be used as arguments to the UNIX commands you specify:

$file	This refers to the file that's displayed in the main window (which may be different from the contents of the window).
$selection	This refers to the contents of the current selection, which can be either text that's selected in a file window or a file that's selected in a folder window.

Here are some examples of how these variables might be used in a **.commanddict** definition:

```
Print Two Up    P    enscript -2r $file
GrepAppkit      A    fgrep -n "$selection" /usr/include/appkit/*.h
```

The first example prints the contents of the file that's displayed in the main window. The second example searches for occurrences of the selected text in the Application Kit header files.

Using a Tags File

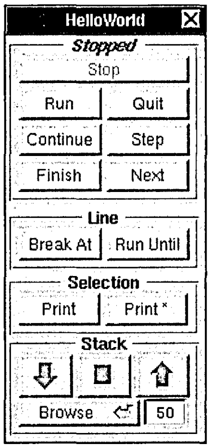
If you're maintaining a large number of files as part of a programming project, you can use Edit's Source command with a **tags** file to quickly locate the definition of an object in that group of files. A **tags** file (which you create with the UNIX **ctags** command) lists the locations of program objects (such as functions, procedures, global variables, and typedefs) that are in a specified group of files.

To locate an object definition, simply select it and choose Source (or choose Source and type the object name in the panel that appears). Edit searches one or more **tags** files for the location of the object definition and then opens the file containing the definition. Normally, Edit searches the **tags** file in the current folder (the folder containing the file in the main window). However, you can specify other **tags** files to be searched either with the Preferences command or by specifying the Tags option when starting up Edit from a shell window.

More information on **tags** files is given in the **ctags** UNIX manual page. For more information on using the Source command, see the command description in "Commands in the Utilities Menu" later in this chapter.

Interacting with the GDB Debugger

A command named Gdb appears in the Edit main menu whenever you execute the GDB **view** command (note that **view** is executed automatically by Project Builder when you ask it to debug a project).



The Gdb command opens the panel shown here, which lets you perform basic debugging operations on a project and its source files. Commands not contained in the Gdb panel can still be accessed by using the GDB debugger directly, as described in Chapter 13.

Since the Gdb command is visible only under certain conditions and has more to do with debugging a project than with editing a document, the Gdb panel is described in the section "Debugging" in Chapter 2, "The Project Builder Application."

Edit Command Reference

The following sections summarize the menus and commands available in Edit.

Commands in the Main Menu

Edit's main menu contains the standard Info, Print, Windows, Services, Hide, and Quit commands. The other commands and the submenus they open are described in the sections that follow. Several standard commands are discussed here only in terms of their particular use in Edit.

Commands in the File Menu

Edit's File menu contains the standard Open, New, Revert to Saved, and Close commands. The other commands are described here.

Command	Description
Save, Save As, Save To, Save All	These are the standard commands for saving the contents of the main window on the disk.
	When you save a file, Edit first moves the contents of the old version to a temporary backup file, which has the same name as the previous file but with a tilde (~) appended to it (for example, the backup file corresponding to **Fruit.m** would be **Fruit.m~**). Next, Edit writes the new version of the file and then it (normally) deletes the backup file. If something happens that prevents Edit from saving the file, however, the backup file remains so you can recover its contents. Or, if you always want the backup file to remain (even after the new version is successfully saved), you can set the "Don't delete backup file" option in the Preferences panel.
	While the file is being saved, "saving:" appears before the file name in the title bar of the window (in the case of small files, it appears only for an instant). Until "saving:" has disappeared, don't use the file (for example, don't try to compile or copy it).
Open Selection	Opens the file or folder currently selected in the main window. Normally, you use this command on a selection in a folder window. However, it also works on selected text in a file window. The selected text must be either a full pathname, or a file name or pathname relative to the current folder (the folder containing the file in the main window).
Open Folder	Displays a panel in which you enter the pathname of a folder to be opened. When you click OK, the folder opens in an Edit folder window. When the panel appears, Edit displays the name of the current folder in the "Folder name" field.

Commands in the Edit Menu

Edit's Edit submenu contains the standard Cut, Copy, Paste, Delete, and Select All commands, plus commands for opening the Link menu and the Find menu described below. Other commands are described here.

Command	Description
Undelete	Reinserts the most recently deleted text, even if the text hasn't been put on the pasteboard. You can insert the deleted text at a new location by positioning the insertion point where you want to insert the text (or selecting text that you want it to replace) and then choosing Undelete.
Spelling	Opens the Spelling Panel for checking the spelling of words in the main window. See "Checking Spelling."
Check Spelling	Has the same effect as clicking Find Next in the Spelling panel—that is, it finds the next word not contained in the spelling dictionary. See "Checking Spelling."

Commands in the Link Menu

Edit's Link menu provides the following commands for working with linked documents. For more information, see "Adding Linked Graphics."

Command	Description
Paste and Link	Pastes a copy of a graphic contained on the pasteboard, but creates a link to the document that the graphic came from, so that future changes to the original graphic will affect the copy in the Edit document as well.
Show Links, Hide Links	Shows (or hides) whether or not graphics are linked by displaying a linked chain around the border of each linked graphic.
Link Inspector	Opens the Link Inspector panel.

Commands in the Find Menu

Edit's Find menu contains the standard Find Panel, Find Next, Find Previous, and Enter Selection commands. Other commands are described here.

Command	Description
Jump to Selection	Scrolls the insertion point or current text selection into view.
Line Range	Opens a panel that identifies by number the line or line range containing the current selection in the main window. If the Character option in this panel is selected instead of the Line option, then the character range is displayed instead of the line range. You can also use the panel to search for a particular line, line range, character, or character range in the main window. Enter a number or a range (a range is two numbers separated by a colon) in the Range field. Click the Select button to select that character, line, or range of the file.

Commands in the Format Menu

The Format menu contains commands for displaying the standard Font and Text menus, as well as Edit-specific Help and Structure menus. Commands on these menus are described later in the sections that follow.

Command	Description
Page Layout	Displays the standard Page Layout panel for choosing among various paper sizes, scaling factors, and orientations for text printed from the main window. When you print text that's displayed in a window, the printed words wrap exactly as they're wrapped on the screen. Therefore, if you change the page layout, the width of the window may also need to be changed in order for the text to print correctly. Changing the page layout doesn't affect the size of the main window, so you'll need to make this adjustment.

Commands in the Font Menu

The Font menu contains the standard Font commands, plus a few additional commands that let you change the font properties of the text displayed in the main window—for example, the Colors command opens the standard Colors panel, which you can use to change the color of the selected text.

In an RTF file, font changes apply to the current selection and are saved when you save the contents of the window. In an ASCII file, font changes are applied to the entire contents of the main window—font changes in non-RTF files aren't saved when you save the contents of the window.

Commands in the Text Menu

Edit's Text menu contains commands that let you change properties of the text displayed in the main window. Some of these commands work only on text in RTF files; use the Make Rich Text command if you want to change the contents of the main window from ASCII to RTF.

Command	**Description**
Align Left, Center, Align Right	These align the text with the left margin (ragged right), center it between both margins, or align it with the right margin (ragged left).
Make Rich Text, Make ASCII	Changes the format of the text in the main window from RTF to ASCII, or vice versa. In an RTF file, font changes and other text properties (such as superscripting and subscripting) can be saved as part of the file and displayed along with the text.
Nest, Unnest	These help you indent blocks of program code. Select the program lines you want to indent and then choose Nest. Each line in the selected program text will be indented the default amount (four characters, unless you've specified a different default value in the Preferences panel or overridden the default when you started up Edit from a shell window). Unnest moves the selected lines the default number of characters to the left, thus counteracting the effect of Nest.

(continued)

Command	Description
Show Ruler, Hide Ruler	Show Ruler displays a ruler at the top of the main window, and the Hide Ruler command removes it. With this ruler you can set margins, tabs, and paragraph indentation. See "Using the Ruler" for details.
Copy Ruler, Paste Ruler	Copy Ruler copies the ruler settings of the paragraph containing the insertion point or the beginning of the current selection, so that you can subsequently paste them with Paste Ruler. It's as though there's a separate pasteboard for the ruler, and Copy Ruler replaces what's already on it, just as Copy does for text. Paste Ruler affects the paragraph containing the insertion point or the current selection. It replaces the paragraph's ruler settings with the last ones you copied with Copy Ruler. If the current selection spans more than one paragraph, Paste Ruler replaces the ruler settings of all the selected paragraphs. These commands don't require the ruler to be showing, and they don't change the contents of the pasteboard.

Commands in the Help Menu

The Help menu provides the following commands, which are used to add or edit Help links. Note that although Help links are designed for use within an application's on-line Help system, they can also be used more generally (for example, the Contents file for the on-line developer release notes contains links to the various release note files). For more information about working with Help links and markers, see "Adding Help Links." For information about adding a Help system to an application you're developing, see Chapter 3.

Command	Description
Insert Link	Inserts a Help link at the insertion point in the main window.
Insert Marker	Inserts a Help marker at the insertion point in the main window.
Show Markers, Hide Markers	Shows (or hides) all the markers in the main window.

Commands in the Structure Menu

The Structure menu provides commands that control whether certain portions of the text in the main window are expanded (that is, visible) or contracted (that is, hidden). These commands are useful for working with files that have a regular multilevel structure, in which the various levels of structure are represented by varying degrees of indentation (for example, an outline or Objective C language source code). See "Contracting and Expanding Text in a File Window" earlier in the chapter for a detailed introduction to this Edit feature.

Command	Description
Contract All, Expand All	These contract or expand all the text in the main window.
Contract Sel, Expand Sel	These contract or expand the selected text in the main window.

Commands in the Utilities Menu

Commands in the Utilities menu perform a variety of functions, such as providing an interface to the UNIX shell and looking up information in a UNIX manual page. There are also two customizable submenus—User Commands and User Pipes—to which you can add commands that you've defined yourself.

Command	Description
Command	Displays a panel in which you specify a UNIX command to be executed. The output of the command appears in a window titled **UNTITLED**, rather than in the main window. Two variables can be used as arguments to the UNIX command you specify: **$file** refers to the file that's displayed in the main window. **$selection** refers to the contents of the current selection, which must be single file specification (wildcards can be used). Normally this will be a file that's selected in a folder window.

(continued)

Command	Description
User Commands	Displays a menu of commands you've defined and saved in a file named **.commanddict** in your home folder. Any changes you make to the **.commanddict** file don't take effect until the next time you start Edit. The **.commanddict** file format is described in "Piping UNIX Output to a File" earlier in this chapter.
Pipe	Works the same as Command, with one important difference: The output of the UNIX command that you specify isn't displayed in another window—instead, the output (including any error messages that might be generated) appears in the main window at the insertion point or in place of the current selection.
User Pipes	Displays a menu that contains pipe commands you've defined and saved in a file named **.pipedict** in your home folder. These commands may be similar to commands you define in the User Commands menu, but the output appears in the main window at the insertion point or in place of the current selection, rather than in a separate window. The **.pipedict** file format is described earlier in "Piping UNIX Output to a File."
Source	Opens the file containing the definition of the program object (such as a function, procedure, global variable, or typedef) selected in the main window. This command searches one or more **tags** files for the location of the object definition and then opens the file containing the definition. Normally, Edit searches the **tags** file in the current folder (the folder containing the file in the main window). However, you can specify other **tags** files to be searched either in the Preferences panel or when starting up Edit from a shell window.

To locate an object definition, select the function name, macro, or other program object in the file you're working in and choose Source. Edit opens the file containing the required information and highlights the first occurrence of the object in the text. If you choose Source without selecting text, Edit displays a panel that prompts you to enter the program object you want defined. If Edit can't locate the object, it informs you that no such **tags** file entry for the object exists. (If this happens, use the Preferences command to make sure that the pathname of the **tags** file listing the location of the object is specified.)

A **tags** file is a file you create with the UNIX **ctags** command. The file lists the locations of specified program objects (such as functions, procedures, global variables, and typedefs). More information on **tags** files is given in the **ctags** UNIX manual page.

Manual	Displays a UNIX manual page in an Edit window. First select the manual page subject in the main window and then choose Manual. If there's no selection, a panel appears prompting you for an entry.
Match	If you select one of a matching pair of delimiters (parentheses, braces, or square brackets) and choose Match, the pair of delimiters and the enclosed text become selected. You can also invoke this command by double-clicking either of the delimiters.

Commands in the Expert Menu

The Expert menu provides the following advanced commands.

Command	**Description**
Update Folder	Updates the contents of the main window, which must be a folder window. Folder windows aren't automatically updated, so this command is useful when files in a folder have been created, deleted, or renamed.
Copy PS	Copies the contents of the main window onto the pasteboard as an Encapsulated PostScript (EPS) image. Once pasted into an application that accepts EPS images, the pasted copy of the text can no longer be edited.
Expansion Dictionary	Opens the Expansion Dictionary panel for managingtext expansion definitions. See "Using Templates" for a complete description of this panel.
Insert Field	Creates a new field in an expansion template. See "Using Templates."
Next Field	Moves the insertion point to the next field in an expansion template. See "Using Templates."
Close Ancestors	Closes all Edit windows associated with each folder that's neither the main window's folder nor one of its subfolders.
Close Descendants	Closes all Edit windows associated with each folder that's a subfolder of the main window's folder. If the main window is a folder window it will remain open, but if the main window is a file window it will be closed as well.

5 The Terminal Application

5 The Terminal Application

Although you can run standard UNIX programs and commands on a NeXTSTEP computer, such programs aren't designed to be run directly from the workspace. Traditional UNIX constructs such as standard input and standard output, which many UNIX-style programs depend on, aren't part of the workspace interface.

To run these programs, you can use the Terminal application. Terminal offers a number of useful features:

- Scrollers let you scroll backward to text that has already disappeared from the window.
- Text can be copied and pasted within a Terminal window, between windows, or to and from other applications that support cutting and pasting, such as Mail and Edit.
- Terminal has a Print command to let you print the contents of a window, and a Find command to let you search for text.
- Terminal's Services menu lets you make interapplication requests, such as defining a word in Digital Webster™ or searching for references in the Digital Librarian™. You can also define your own Terminal services for use in other applications.
- Terminal's Preferences command allows you to change the size, title bar text, emulation characteristics, and font properties of one or more Terminal windows.
- Terminal provides strict VT100™ terminal emulation. Every UNIX program or utility you run (such as Emacs or **vi**) should work as intended.

The rest of this chapter describes Terminal in more detail.

Introduction to Terminal

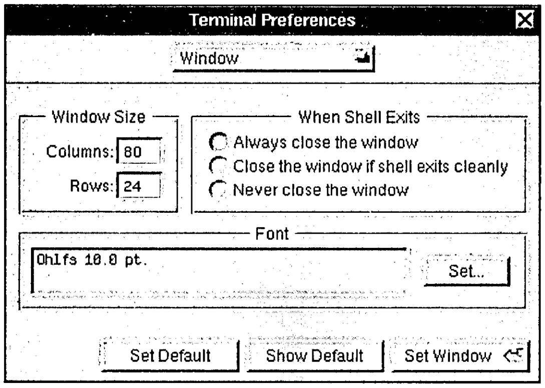
A UNIX shell is a program that functions as an intermediary between you and the UNIX operating system. As the shell runs, it prompts you for commands, interprets what you type, and passes the commands to the operating system for execution. For more information about the two most common UNIX shells, the Bourne Shell and the C Shell, see their UNIX manual pages (**sh**(1) and **csh**(1)).

You start Terminal (located in **/NextApps**) from the workspace as you would any other application, by double-clicking its icon in the workspace or by using the Workspace Manager's Preferences command to make Terminal start up when you log in. When Terminal starts up, it will (if configured to do so) create a new Terminal window using the default Preferences settings. You can create additional Terminal windows as you need them using the New Shell command. To change a window's characteristics, select the appropriate settings in the Preferences panel as described in the following section.

Setting Preferences

The Preferences command in the Info menu displays the Preferences panel, shown below. The Preferences panel lets you change values and set new default values for various Terminal options. For example, you can set the font properties of a particular window, or specify different default font properties to be used for new windows. This section describes the various preferences. The illustrations show the settings you start out with the first time you use the Terminal application. As you click in shell windows, the Preferences panel shows the settings for the main window.

Enter values and click buttons to specify new preferences, as described below. You may need to click Set Window to set the new preferences (or, click Set Default to make the new settings be the default settings or Show Default to show the currently defined default settings). New settings remain in effect until you change them. However, some Preferences settings affect only new windows but others affect existing Terminal windows as well. (Specifically, when no buttons appear at the bottom of the panel, settings are global and apply to all shell windows.)



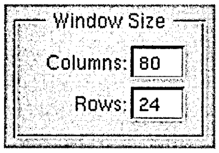
Preferences options are divided into the following seven groups:

- Window preferences
- Title Bar preferences
- VT100 Emulation preferences
- Display preferences
- Activity Monitor preferences
- Shell preferences
- Startup preferences

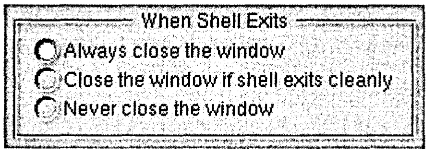
Each group of options is displayed in its own view in the Preferences panel. Select the view you want by clicking the button labeled Window at the top of the panel and dragging.

Window Preferences

You can use the Window Preferences panel to set the size and font of one or more Terminal windows. If you click Set Window, the settings are applied to the Terminal window that's currently the main window. If you want the settings to apply to new windows, click Set Default.

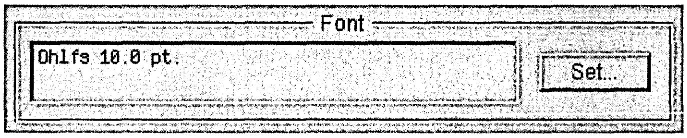


The Columns and Rows fields specify values for the number of columns and rows. Even after setting the number of columns and rows, you can still resize the window, thereby changing the number of columns and rows for that window.


When Shell Exits

( ) Always close the window
( ) Close the window if shell exits cleanly
( ) Never close the window

This field lets you specify what you want to have happen to a window when the shell running in it exits. In some special situations, a window might not obey the default setting. For example, double-clicking a command in the workspace results in a window that stays open even after the command finishes executing.


Font

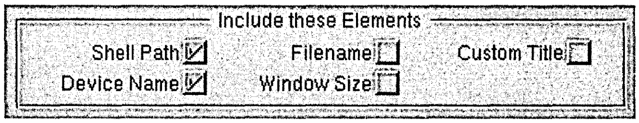
Ohlfs 10.0 pt.

Use the Font field to specify a font for one or more Terminal windows as follows:

- Click the Set button to open the Font panel.
- In the Font panel, select a font (note that only fixed-width fonts are listed) and font size. Click the Set button in the Font panel to enter the settings in the Font field of the Preferences panel.
- Click Set Window in the Preferences panel to set the font for just the main window, or click Set Default to make this font the default for new windows.

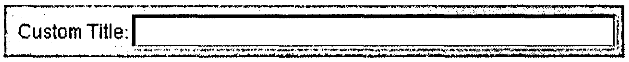
Title Bar Preferences

You can use the Title Bar view of the Preferences panel to configure the title bar of one or more Terminal windows. If you click Set Window, the new settings you specify are applied to the Terminal window that's currently the main window. If you want the settings to apply to new windows you create, click Set Default.

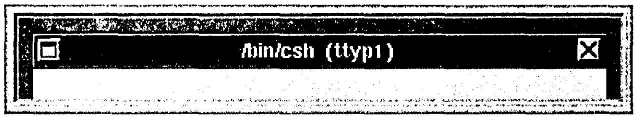

Include these Elements

Shell Path ✓ Filename ☐ Custom Title ☐
Device Name ✓ Window Size ☐

This field provides a number of elements that you can include in the title bar of Terminal windows, including a "custom title" element that you define yourself. Any combination of elements can be used. If no elements are selected, the title that's used is simply **Terminal**.


Custom Title:

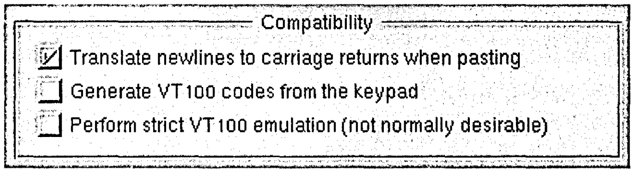
To specify a custom title, enter it in the Custom Title field. The custom title is used in the title bar, however, only if you click the Custom Title box in the Elements field above.


/bin/csh (ttyp1)

As you experiment with various combinations of elements, the sample title bar displayed in the Preferences panel is updated to show the effect of the current settings.

VT100 Emulation Preferences

The VT100 Emulation view is used to set the VT100 characteristics of Terminal windows.


Compatibility

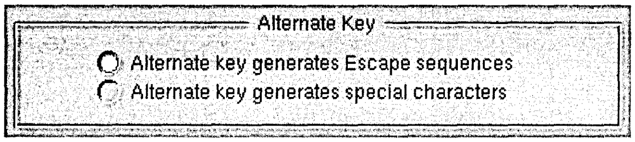
- [✓] Translate newlines to carriage returns when pasting
- [ ] Generate VT100 codes from the keypad
- [ ] Perform strict VT100 emulation (not normally desirable)

"Translate newlines to carriage returns when pasting" should normally be checked. It's required by some other operating systems, and it works correctly for most UNIX programs.

If "Generate VT100 codes from the keypad" is checked, the keys on the numeric keypad generate VT100 keypad sequences. Otherwise, the keys on the numeric keypad generate the characters shown on the keys. Holding down the Alternate key while pressing a key on the numeric keypad toggles the interpretation temporarily.

If "Perform strict VT100 emulation" is checked, some additional (and normally undesirable) aspects of VT100 emulation are strictly enforced:

- If you type a Delete character at the left edge of a Terminal window, the command-line cursor won't wrap around to the end of the previous line. This may make it difficult to edit long command lines that wrap.
- Strict DECCOLM handling is enforced. Otherwise, the DECCOLM escape code to change the window's size is obeyed only if the new size is larger than the old size.
- The + key on the numeric keypad generates a comma (,) character.

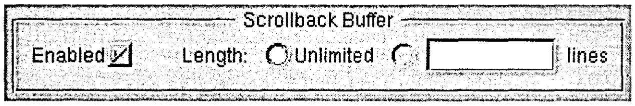

Alternate Key
Alternate key generates Escape sequences
Alternate key generates special characters

When "Alternate key generates Escape sequences" is selected, typing a character while you hold down the Alternate key causes a two-character sequence to be generated—an Escape character followed by the character you typed. (This is useful when running Emacs, so you can use the Alternate key as a Meta key). Click "Alternate key generates special characters" if you want Alternate key combinations to generate a single character with the high bit set.

Note: If necessary, you can specify a character other than Escape as the first character in a two-character sequence. To do so, use the **dwrite** shell command to set the value of the Terminal **Meta** variable to the decimal value of the desired character.

Display Preferences

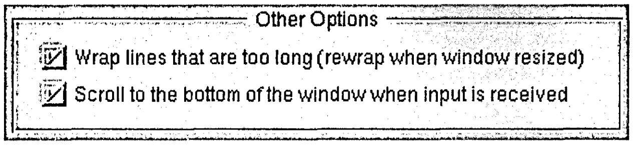
The Display view of the Preferences panel is used to set various display characteristics of one or more Terminal windows. If you click Set Window, the new settings you specify are applied to the Terminal window that's currently the main window. If you want the settings to apply to new windows you create, click Set Default.



If the Enabled box is checked, windows retain text that scrolls off the top of the window in a scrollback buffer, allowing text that's scrolled off the window to be scrolled back into view, copied, or printed. Otherwise, text that scrolls off the top of the window can't be retrieved.

If you enable the scrollback buffer, you can choose to let it grow without limit or you can specify the maximum number of lines that you want saved. Whichever you choose, you can use the Edit menu's Clear Buffer command at any time to clear the buffer.

Although it's often useful, the scrollback buffer adds to the amount of memory that's used by the Terminal program, and is unnecessary in some Terminal windows (for example, one that's running a text editor such as Emacs rather than a UNIX shell).



Other Options

- [✓] Wrap lines that are too long (rewrap when window resized)
- [✓] Scroll to the bottom of the window when input is received

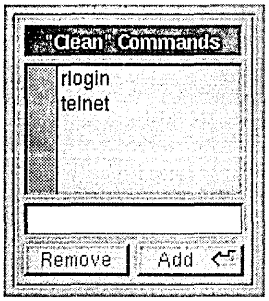
If the "Wrap lines that are too long" box is checked, characters that would extend beyond the right edge of the window wrap around to the beginning of the following line. Otherwise, each line of text occupies only one line in the window—the last character that fits on a line gets overwritten by subsequent characters that appear on that line.

If the "Scroll to the bottom of the window when input is received" box is checked, typing in the Terminal window causes the window to scroll to the end of the buffer and display the insertion point (of course, if the insertion point happens to be already visible and positioned at the end of the buffer, no scrolling occurs). Otherwise, typing never causes the window to scroll automatically.

Activity Monitor Preferences

Normally, Terminal tries to determine whether your Terminal windows are in active use (busy) by keeping tabs on the processes inside them. If Terminal thinks something interesting is going on inside a window, it marks the window with a broken X. As with unsaved document windows in other applications, you'll be prompted for confirmation before closing a busy window or quitting Terminal when there are busy windows.)

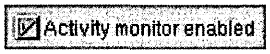
To determine whether a window is clean (not busy), Terminal looks at information about processes it considers relevant. For example, Terminal considers shells and a few other processes such as **su** to be innocuous and in general will not mark windows busy on account of them. Occasionally Terminal may be wrong about whether a window is clean or not.



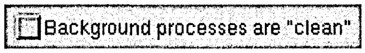
Clean Commands

rlogin
telnet

You can designate additional clean command names in the Clean Commands list (likely candidates are **rlogin** and **telnet**, shown here). Commands you specify in this list aren't used in determining whether a window is busy or clean.


☑ Activity monitor enabled

Click this button if you want to enable or disable activity monitoring. When activity monitoring is off, Terminal always asks for confirmation before letting you quit.

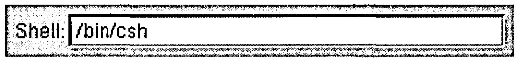

☐ Background processes are "clean"

Click this button if you want to specify whether or not background processes are considered relevant in determing whether a window is clean. For example, a window running a background process could be considered clean, as long as the process is running happily in the background. (The current process or one that you've explicitly suspended with Control-z will always cause the window to be classified as busy.)

Note: Terminal can't respond to processes running on other machines, so you shouldn't rely on Terminal's process monitor when logged into a remote system.

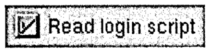
Shell Preferences

The Shell view of the Preferences panel is used to specify a shell or other program to be run in Terminal windows.


Shell: /bin/csh

Use the Shell field to specify the absolute pathname of a shell or program to run on startup. Possible values include **/bin/csh**, **/bin/sh**, **/bin/gdb**, **/usr/bin/emacs**, and **/usr/ucb/vi**.

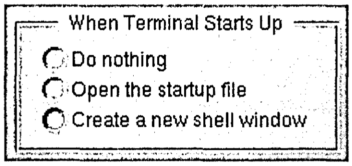
Note: You must press the Return key after entering the pathname in order for the new value to be set.


☑ Read login script

If the "Read login script" box is checked (and you're using **csh**), Terminal runs your **.login** file for each new Terminal window you open. Otherwise, the **.login** file is ignored.

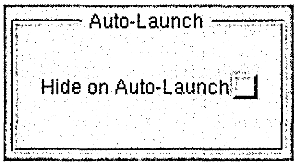
Startup Preferences

The Startup view of the Preferences panel lets you specify what happens when Terminal starts up.

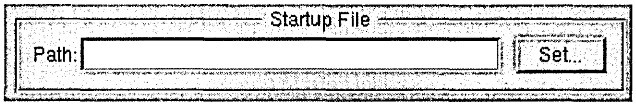


When Terminal starts up you can have it do nothing (that is, create no windows), create one new shell window, or open a startup file (that is, a configuration file that specifies a collection of windows to open). If you select "Open the startup file," you need to make sure a startup file is specified in the Startup File portion of the panel, described below.

For information about how to create a startup file, see "Saving a Terminal Configuration for Later Use."



Click here if you auto-launch the Terminal application and want it to be hidden initially. This button has no effect if you don't auto-launch Terminal.

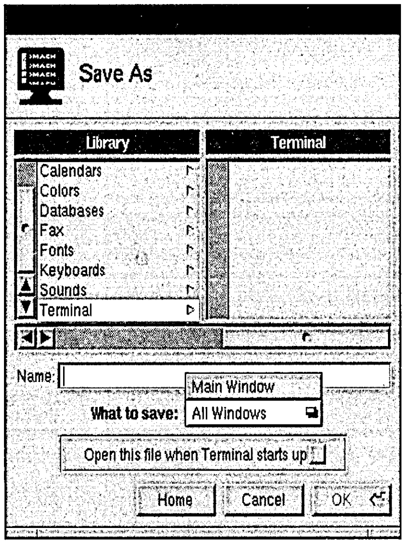


Although you can have any number of Terminal configuration files in your **~/Library/Terminal** directory, you can specify only one as the startup file. To specify a particular Terminal configuration file as the startup file, type its pathname in the Path field or click Set to open an Open panel in which to select the pathname.

Note: The pathname you enter must be an absolute pathname beginning with a slash (/); characters such as ~ won't work.

Saving a Terminal Configuration for Later Use

Information about a window or set of windows can be saved to a file, allowing you to save your preferred configurations for later use. Everything about each window is saved except the contents of the scrollback buffer—this includes the shell, the size and location of the window on the screen, the title bar and font characteristics, and whether or not the window is miniaturized. To save a configuration, choose Save (or Save As) in the Shell menu. Terminal appends a **.term** file extension to the file name you specify. Since Terminal looks for configuration files in your **~/Library/Terminal** directory, this is where you should save them.



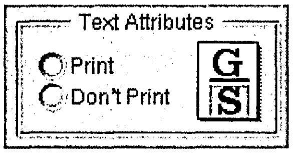
When you first save the configuration (or whenever you choose Save As), you can choose whether you want just the main window or all windows saved to the file.

Once a window is associated with a file, you can use the Save command to flush the settings out again without seeing a Save Panel, just as with other documents. However, if more than one window belongs with that file, all the relevant windows will be resaved (the menu item indicates this by changing to Save Set). This allows you to open your favorite set of files, rearrange the windows, then just choose Save to save them all back into the file. There is no way to select a subset of the currently open windows to go into a new file.

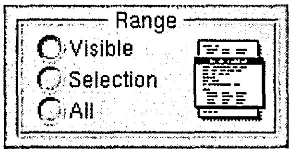
To open a configuration file, choose Open in the Shell menu. To have a configuration file open automatically each time you start up Terminal, either check the box in the Save or Save As panel, or specify the filename in the Startup view of the Preferences panel.

Printing the Contents of a Terminal Window

To print all or part of the text in a Terminal window, open the Print panel by choosing Print in the main menu. Terminal's Print panel is similar to the standard NeXTSTEP Print panel, but there are two options not contained on the standard panel.



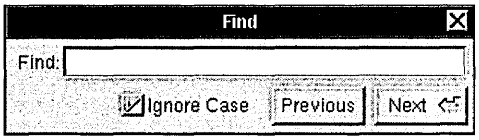
Click the Print button or the Don't Print button to specify whether or not text attributes (underlining and highlighting) are included in the printed output. Print indicates that text attributes appear in the output; Don't Print indicates that the text attributes won't appear.



Click one of the three Range buttons to specify the range of text to be printed. All indicates that the entire contents of the scrollback buffer should be printed. Selection indicates that the selected text, whether visible or not, should be printed. Visible indicates that the text that's visible in the window should be printed.

Finding Text in a Terminal Window

The Find panel lets you search for text in the main Terminal window. To open the Find panel, choose Find Panel in the Find menu.



The Find panel locates the next occurrence of a specified string, and can search either forwards or backwards. In the Find field, enter the string to search for. The controls in the Find panel have these effects:

Control	**Effect**
Next	Selects the first occurrence of the Find string following the current selection or insertion point. (Pressing the Return key has the same effect, but with one difference: If you've used the keyboard alternative to display the panel, pressing Return causes the panel to disappear instead of remaining on the screen.)
Previous	Selects the first occurrence of the Find string, searching backward from the current selection or insertion point.
Ignore Case	Makes the find operation case-insensitive (that is, capitalization is ignored when determining a match). If this box is unchecked, the search is case-sensitive.

If the end of the text is reached, Find continues searching from the beginning (conversely, when searching backward, if the beginning of the text is reached, Find continues searching from the end).

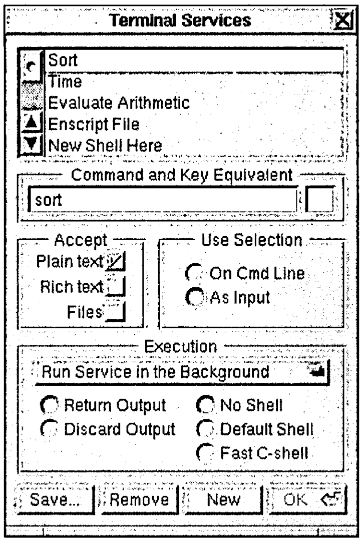
If no instance of the Find string is located, Terminal beeps and the message "Not Found" appears in the Find panel.

Commands in the Find menu (which is in the Edit menu) provide alternatives and shortcuts to using the Find panel. There's also a Jump to Selection command for scrolling the insertion point into view. For more information, see "Terminal Command Reference" at the end of this chapter.

Defining Services for Use in Other Applications

Although by default Terminal doesn't make services available to other applications via the Services menu, Terminal does contain a Terminal Services panel that you can use to define any services you want Terminal to provide.

To open the Terminal Services panel, choose Terminal Services in the Info menu. If no Terminal services are defined, you'll see a panel asking if you want to load a set of example services. You may find it useful to load and examine this set of examples, and then remove any you don't want to keep.



Currently defined services are listed in the list at the top of the panel. You can add new services, as well as redefine or delete existing commands.

- To add a service, click the New button. A new entry named New Service #1 is added to the service list. Type the name that you'd like to appear in the Services menu, and then configure the service using the controls in the Terminal Services panel. When you're done, click OK.

- To delete a service, select it and click the Remove button.

- To modify a service definition, select its name and then redefine the service using the controls on the Terminal Services panel. When you're done, click OK.

The Accept field lets you specify what type of data the service accepts. Click one or more of these buttons as appropriate.

The Use Selection field lets you specify whether the selected text should be used as a command-line argument, or as input to the service. Click one or the other as appropriate.

The Execution field lets you specify various options that affect the execution of the service, such as whether the output is returned or discarded.

When defining the service, you can use the tokens **%s** and **%p** to refer to the locations where the selection and prompted input are inserted, respectively. Prompted input isn't requested unless **%p** appears in the definition.

Terminal Command Reference

The following sections summarize the menus and commands available in Terminal.

Commands in the Main Menu

Terminal's main menu contains the standard Print, Windows, Services, Hide, and Quit commands. The other commands and the submenus they open are described in the sections that follow. Several standard commands are discussed here only in terms of their particular use in Terminal.

Commands in the Info Menu

Terminal's Info menu provides the standard Info Panel command, plus the following commands.

Command	Description
Preferences	Opens the Preferences panel. See "Setting Preferences."
Terminal Services	Opens the Terminal Services panel. See "Defining Services for Use in Other Applications."

Commands in the Shell Menu

Terminal's Shell menu provides the following commands.

Command	Description
Open	Opens an existing shell window or set of shell windows that have previously been saved in a file using the Save (or Save As) command.
New	Opens a new shell window, using the default settings.

Run Command	Displays a panel in which you enter a UNIX command to be run. The command is run in a new Terminal window. (The command is displayed as the title of the window; when the process running in the window has completed, the title changes to "Dead Terminal.")
Save, Save As	Saves a window or set of windows to a file, allowing you to save and reuse your preferred configurations. See "Saving a Terminal Configuration for Later Use."
Set Title	Displays a panel for you to edit and set the current title of the window. The Preferences panel allows greater control over this—you can combine your own text with Terminal's automatically updated information. See "Title Bar Preferences" for more information.
Steal Keys	Allows you to effectively debug an application from a shell window in which the GNU debugger is running. The debugging process frequently involves alternately activating Terminal (to type debugger commands) and the other application (to test the application being debugged). However, clicking to alternatively activate and deactivate the application being debugged causes the application to change its state in unpredictable ways. To let you get around this problem, the Steal Keys command puts Terminal in a special debugging mode. In this mode, Terminal can be activated or deactivated simply by moving the cursor into or out of the Terminal shell window. Therefore, you can easily activate Terminal whenever you want to type a debugger command, without clicking and thus affecting the state of the application you're debugging. When you're ready to exit debugging mode, click in the Terminal window to make the Terminal main menu reappear, and then choose this command again (its name will have changed to Yield Keys).
Page Layout	Displays the standard Page Layout panel, which lets you choose among various paper sizes, scaling factors, and orientations for text printed from the main window.

Commands in the Edit Menu

Terminal's Edit menu provides the standard editing and text-searching commands, which can be used for finding and editing text in a Terminal window.

Command	Description
Cut, Copy, Paste	These commands let you copy or move text, either between Terminal windows or between a Terminal window and another window that supports copying and pasting. To duplicate text, select the text and choose Copy. To insert the most recently cut or copied text at the Terminal window's command-line cursor location, choose Paste. Copy puts a copy of the selected text onto the pasteboard, from where it can be pasted with the Paste command. The pasteboard holds just one selection; each Copy operation overwrites the previous contents of the pasteboard. **Note:** Cut is always disabled. The only way to remove text from a Terminal window is to use the Clear Buffer command.
Find	Displays a menu that contains commands for finding text, as described below in "Commands in the Find Menu."
Clear Buffer	Removes text from the scrollback buffer, leaving just the current command line.
Select All	Selects all the text in the main window. This is useful, for example, when you want to copy the entire range of text to another application, such as Edit.

Commands in the Find Menu

The Find menu contains commands that let you search for text in the main Terminal window.

Command	Description
Find Panel	Opens the Find panel, which allows you to locate the next occurrence of a specified string. For more information, see "Finding Text in a Terminal Window."
Find Next, Find Previous	These are the standard Find menu commands. The Find Next command performs the same function as the Next button in the Find panel, and Find Previous is the same as the Find panel's Previous button.
Enter Selection	Copies the selected text in the main window into the Find panel's Find field, even if the Find panel isn't open or the key window.
Jump to Selection	When the insertion point or current text selection isn't showing in the main window, the Jump to Selection command scrolls it into view. If there's no insertion point or current text selection, this command scrolls to the end of the buffer. **Note:** Clicking in a Terminal window positions the insertion point where you clicked. However, the insertion point isn't visible since it's not possible to perform any copy or paste operation on it. This may cause some confusion, since the Jump to Selection command may sometimes jump to a location that doesn't appear to have any selected text associated with it.

Commands in the Font Menu

The Font menu contains the standard Font menu commands described in the *User's Reference Manual*. However, these commands apply to the entire contents of the Terminal window, not just to selected text.

Command	Description
Font Panel	Displays the standard Font panel, which lets you choose among various fonts, typefaces, and font sizes. However, only fixed-width fonts, such as Courier and Ohlfs, can be used in Terminal. Also note that Ohlfs is strictly a screen font—text displayed in Ohlfs prints as Courier instead.
Bold, Italic	Makes the text in the main Terminal window become bold or italic.
Larger, Smaller	Makes the text in the main Terminal window become larger or smaller.
Copy Font, Paste Font	Copy Font copies the font settings of the main window, so that you can paste them into another window with the Paste Font command.

6 The Icon Builder Application

6 The Icon Builder Application

The Icon Builder application is a simple yet effective tool—either alone or in combination with a more powerful drawing application—for creating application icons. Although Icon Builder itself isn't intended to be a full-featured drawing application, it offers not only integration with other drawing applications, but also the ability to create and edit multiple-icon documents.

You can start Icon Builder (located in **/NextDeveloper/Apps**) from the workspace as you would any other application, by double-clicking its icon in the workspace. When Icon Builder starts up, it displays a panel of tools used to edit icon documents.

Creating, Opening, and Saving Documents

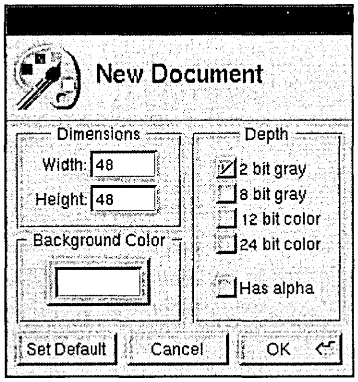
When Icon Builder starts up, it creates one new Icon Builder window using the default Preferences settings. You can create additional Icon Builder windows as you need them, as described in the following section.

Creating a New Document

To create a new document, choose the New command in the Document menu. This creates a document with the default attributes (typically, the document contains a single 48-pixel by 48-pixel, non-alpha, 2-bit gray image with a white background). You can change the attributes of the document after it's been created, as described later in the section "Changing the Attributes of a Document."

To create a new document with nondefault attributes:

1. Bring up the New Document panel by choosing the New Layout command in the Document menu.


New Document

Dimensions
Width: 48
Height: 48

Background Color

Depth
[✓] 2 bit gray
☐ 8 bit gray
☐ 12 bit color
☐ 24 bit color

☐ Has alpha

2. Set options in the New Document panel, and then click OK to create a new document. If you want to change the default attributes for all documents created with the New command, click Set Default instead.

For example, if you want to create an icon for use on both color and black and white displays, you would check the "2 bit gray" box as well as the "12 bit color" box ("8 bit gray" and "24 bit color" could also be used). Check the "Has alpha" box if you'll be using alpha. To change the background color, pick a color in the Colors panel (accessed in the Tools menu) and drag the color into the Background Color color well.

Opening an Existing Document

To open an existing document, choose the Open command in the Document menu and use the Open panel to find the document.

The document you open may be an icon you're working on, or it may simply contain an image that you want to copy a selection from in order to paste it into another document. In addition to TIFF files, Icon Builder can open GIF and EPS files.

Saving a Document

To save a document to a file, choose the Save command in the Document menu. If the document hasn't been saved yet, a Save panel appears prompting you to specify a name and location for the file.

Even if the file you're saving was opened as something other than a TIFF file (a GIF file, for instance), it will be saved as a TIFF file.

Icon Builder saves TIFF files in uncompressed format, so before making the file part of your application project, you should use the **tiffutil** utility to compress the file. See the **tiffutil**(1) UNIX manual page for more information.

Editing Icon Documents

This section describes various ways to edit an icon document, including the set of Icon Builder tools and an inspector for fine-tuning those tools. Other editing techniques described involve zooming in and out, changing the attributes of a document, and working with multiple-icon documents.

Standard cut, copy, and paste techniques can also be used, although these aren't described here.

Using Icon Builder Tools

A variety of drawing tools are available and accessible from the Tools panel, which appears automatically when you start Icon Builder. If you close or misplace the panel, you can retrieve it by choosing the Tools command in the Tools menu.

To use a tool, select it by clicking its icon in the Tools panel. Once you've selected a tool, use it to edit the contents of the document window.

Note: When using the Tools panel, you should have the Colors panel open as well. All the drawing tools draw using the color (or shade of gray) that's currently displayed in the Colors panel. You can also use the Colors panel to specify the degree of alpha coverage (that is, opacity), as well as whether or not painting is done in overlay mode.

The following paragraphs describe the tools on the Tools panel.

The Brush tool is useful for filling in large areas with a particular color. Click once to deposit a brushful of color, or click and drag to cover a larger surface area.

The Line tool draws straight lines. Click to mark the start point, and drag to the end point. The line you see being drawn as you drag is only for guidance—the final line is drawn only when you release the mouse.

The Oval tool draws circles and ovals. Click and drag to determine the position, size, and shape. It's hard to predict the start and end point with accuracy, so you may want to use another document window as a scratch area and then copy and paste the oval once you're satisfied with it.

The PaintBucket tool changes the color of a contiguous, identically colored group of pixels. The color they're changed to is the color that's currently in the Colors panel. Before using the PaintBucket tool, you may want to use the ObeseBits panel to be sure that all the pixels are in fact identical in color—if minor gradations in color are used to achieve the appearance of a particular shade, the new color won't spread from pixel to pixel.

The Pencil tool draws freehand lines. Click to start the line, and drag to indicate the path of the line. Unlike the Line tool, the Pencil tool draws the final line as you drag. If you don't like the result, use the Undo command in the Edit menu to undo it.

The Rectangle tool draws squares and rectangles. Click to position a corner point, and then drag in any direction to form the rectangle.

The Selection tool selects a rectangular area for further editing. For example, after selecting an area you might go on to copy the selection to the pasteboard, or even delete the selection.

The Text tool is used to add text to an image. If you select the Text tool and then click the cursor in a document window, the contents of the TextTool inspector (by default, the word Text—probably not what you want in your icon!) are copied to the cursor location. As long as you don't release the mouse button you can drag the text to reposition it, but once you let go the text becomes fixed in that position.

In order to use the Text tool effectively, you should first enter the desired text in the TextTool inspector. Set the font attributes and font size as you wish. Then use the Text tool to insert the text in the document window, or—to be on the safe side—insert the text first in a temporary scratch document, and then cut and paste the text into the document window.

Other tools besides the Text tool have default attributes that you can change using the Tools Inspector, as described in the next section.

Using the Tools Inspector

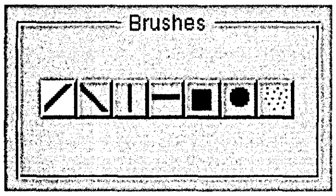
The Tools inspector is a panel that gives you greater control over the characteristics of the tools available in the Tools panel.

To bring up the Tools inspector, choose the Inspector command from the Tools menu. To display the inspector for a particular tool, click the tool's icon in the Tools panel.

Note: There is no inspector available for the PaintBucket tool.

The Brush Inspector

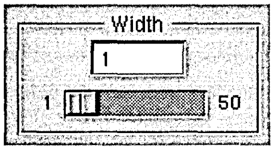
The Brush inspector appears in the Inspector panel when you select the Brush tool in the Tools panel.



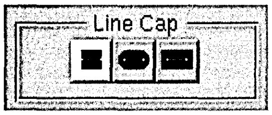
Click a button to change the shape and orientation of the brushstrokes you make.

The Line Inspector

The Line inspector appears in the Inspector panel when you select the Line tool in the Tools panel. You can use this inspector to change the width and end shape of the lines you draw.



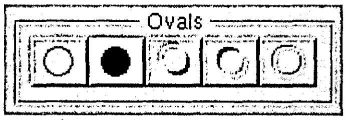
Use the slider or the text field to set the line width to any value between 1 and 50 pixels.



Click one of the three Line Cap buttons to determine how the ends of lines are drawn. The setting becomes less critical as the line width decreases—a one-pixel line is drawn the same no matter what style of line cap is selected.

The Oval Inspector

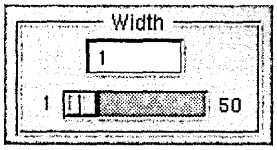
The Oval inspector appears in the Inspector panel when you select the Oval tool in the Tools panel.



Click one of the five buttons to change the appearance of the circles and ovals you draw.

The Pencil Inspector

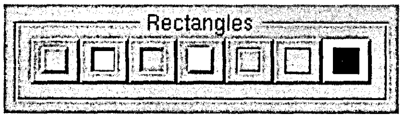
The Pencil inspector appears in the Inspector panel when you select the Pencil tool in the Tools panel. You can use this inspector to change the width of the freehand lines you draw.



Use the slider or the text field to set the line width to any value between 1 and 50 pixels. Thicker lines cause the drawing speed to decrease, so you may need to move the mouse more slowly in order for the drawing process to keep up with it.

The Rectangle Inspector

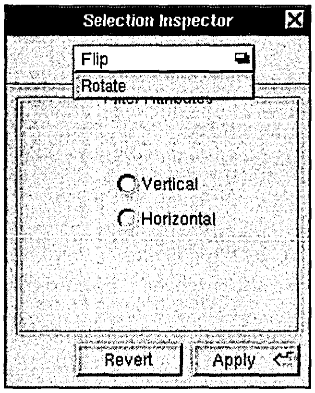
The Rectangle inspector appears in the Inspector panel when you select the Rectangle tool in the Tools panel. You can use this inspector to change the appearance of the squares and rectangles you draw.



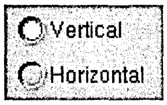
Click one of the seven buttons to change the appearance of the squares and rectangles you draw.

The Selection Inspector

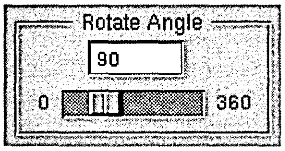
The Selection inspector appears in the Inspector panel when you select the Selection tool in the Tools panel. You can use this inspector to change the orientation of (that is, flip or rotate) the current selection.



Choose Flip or Rotate in the pop-up list at the top of the panel. The available options vary depending on which you choose.



If you choose Flip, you'll see these Flip filter attributes. Click either Vertical or Horizontal to indicate the direction in which you want the selection to be flipped.

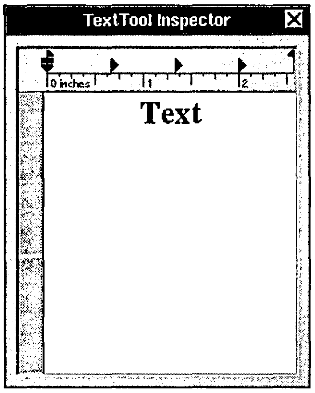


If you choose Rotate, you'll see these Rotate filter attributes. Specify a value between 0 and 360 using either the slider or the text field. This value represents the number of degrees the selection will be flipped in a clockwise direction.

Once you've selected the options, click Apply to flip or rotate the selection. If you don't like the results, click Revert to return the selection to its former orientation.

The TextTool Inspector

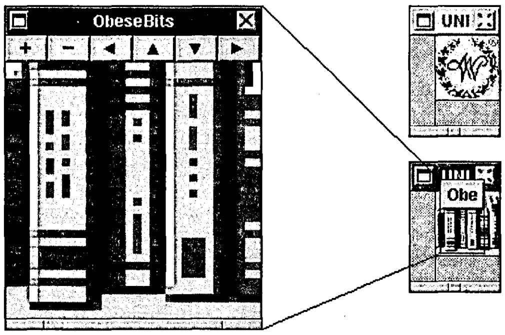
The TextTool inspector appears in the Inspector panel when you select the Text tool in the Tools panel. You use this inspector to input the text to be inserted in the document window, as well as to change the font attributes and formatting of the text prior to inserting it.



Type the text you want to insert in the document window. Use the Font panel to set the font attributes and size of the text. (You may also want to use commands in the Text menu to format the text.) Then select the Text tool and click in the document window to insert the text in the document.

Zooming In on a Document

When you're doing detail work on an image that's only 48 pixels across, you may find yourself wishing you had a magnifying glass. If you start feeling this way, choose the ObeseBits command in the Tools menu to bring up the ObeseBits panel.



This panel magnifies the image in the main window, and lets you zoom in or out (that is, increase or decrease the degree of magnification). The buttons along the top of the panel give you the control you need—use the plus and minus buttons to zoom in or out, and the arrow buttons to change the portion of the image being displayed.

There are actually two ObeseBits panels, as shown in the figure. The big panel is for editing; the small panel sits over the document window and shows the exact portion of the image that's contained in the big panel. You can drag the small panel by its title bar, thereby changing the portion of the image being displayed in the panel.

The ObeseBits panel associates itself with whatever window is the main window. Clicking the document containing the Webster icon in the figure, for example, would cause the small ObeseBits panel to jump to that document window. The contents of the big panel would change accordingly.

Note: Although the drawing tools in the Tools panel can be used directly in the ObeseBits panel, the result isn't always intuitive. For example, the size of the Brush cursor doesn't accurately represent the brush size that's used when stroking the brush. Use the Undo command in the Edit menu to undo any changes that you regret making.

Changing the Attributes of a Document

After you've created a document or opened an existing document, you may find it necessary to change its size, format, or other characteristics. For example, you might decide to add alpha to a document that doesn't have it.

To make such changes, first click the document window to make sure it's the main window. Then open the Document Layout panel by choosing the Document Layout command in the Format menu.

The Document Layout panel is identical to the New Document layout panel that was described earlier. The only difference is that this panel is used to change the attributes of an existing document, rather than determine the attributes of a new document.

Note: If you set new default attributes in the Document Layout panel, these become the default attributes for the New Document layout panel as well.

Working with Multiple-Icon Documents

One Icon Builder document (that is, one TIFF file) can contain more than one icon. This is typically the case, for example, when you want to have one icon for color monitors and another for grayscale monitors—if the two icons are in the same TIFF file, the appropriate icon is displayed automatically on each type of monitor, without any work on your part.

To create a multiple-icon document (or to change an existing single-icon document to a multiple-icon document), select the desired depth settings in the New Document panel (or the Document Layout panel). Then click OK.

Use the pop-up list that appears in the lower right corner of a multiple-icon document window to access the various icons.

If you create a multiple-icon document, remember that you have to edit each icon. When you save the document, all the icons in the document are saved—not just the one that's currently visible in the document window.

Icon Builder Command Reference

This section describes the application-specific menus and commands available in Icon Builder. For descriptions of standard menus and commands, see the *User's Guide.*

Commands in the Main Menu

Icon Builder's main menu contains the standard Edit, Windows, Print, Services, Hide, and Quit commands. The Document, Format, and Tools commands display submenus that are described in the following sections.

Document	Displays a menu of commands for creating, opening, and saving document windows. See "Commands in the Document Menu."
Format	Contains commands for opening the standard Font and Text menus, plus the Document Layout command for specifying the layout of the document in the main window. See "Commands in the Format Menu."
Tools	Contains commands for opening a panel containing the tools available for use in creating an icon. See "Commands in the Tools Menu."

Commands in the Document Menu

Icon Builder's Document menu provides the following commands.

Command	Description
Open	Opens an existing document window. See "Opening an Existing Document."
New	Opens a new document window using the default attributes. See "Creating a New Document."
New Layout	Displays a panel that lets you change the default attributes used in creating a new document. See "Creating a New Document."
Save, Save As	Saves a document (consisting of one or more TIFF images) to a TIFF file. See "Saving a Document."
Revert to Saved	Undoes the changes that have been made since the last time the document was saved.

Commands in the Format Menu

Icon Builder's Format menu contains menus of standard font and text commands, which can be used to affect the appearance of text used in Icon Builder.

Command	Description
Font	Opens a menu of standard Font commands, which you use to set the font characteristics of the selected text in the TextTool inspector.
Text	Opens a menu of standard Text commands, which you use to set the attributes of the selected text in the TextTool inspector.
Document Layout	Displays the Document Layout panel, which you use to change the attributes of a document window. See "Changing the Attributes of a Document."

Commands in the Tools Menu

The Tools menu contains commands for accessing the primary tools provided in Icon Builder.

Command	Description
Inspector	Opens the Inspector panel, which you use to change the appearance and behavior of the available tools. See "Using the Tools Inspector."
Tools	Opens the Tools panel, which you use to select among available tools. See "Using Icon Builder Tools."
Colors	Opens the standard Colors panel, which you use to choose color or grayscale values.
Obese Bits	Opens the ObeseBits panel, which you use to magnify the contents of a document window. See "Zooming In on a Document."
Load Tool	Opens the Load Tool panel, which you use to load a nonnative tool located somewhere on the file system.

7 The DBModeler Application

7 The DBModeler Application

With the DBModeler application, you can quickly and easily build data models based on the structure of your relational databases. DBModeler derives information about the database's structure or schema through an adaptor, and then uses that information to generate a default model (in essence, the DBModeler translates a database's table-and-column view of the data elements into logical hierarchies of related elements). You can then use DBModeler to enhance the structural foundation provided by the default model—for example, by specifying relationships between data elements and between tables.

The resulting model of the data structure more closely represents the real world relationships that exist between data elements. It also allows navigational access across data source entities.

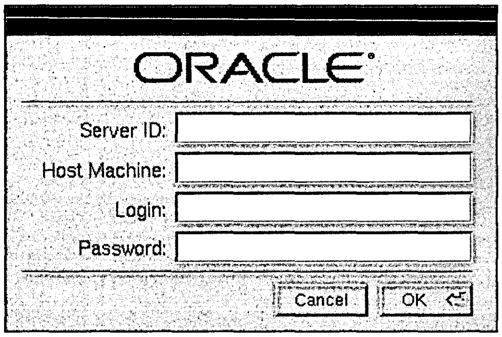
The models you create can then be used in Interface Builder (and by DBDatabase objects) to construct applications that access the data in the source databases. By incorporating models into the development process, you no longer have to worry about the semantics of accessing specific data elements, or constantly reconstructing these relationships for every new application as it is built.

Note: DBModeler is designed for application developers who have a working knowledge of object-oriented application development and database systems. This chapter presupposes that you are already familiar with the high-level dynamics of the Database Kit, and with your external source of data (that is, the database you'll be creating the model from). It's also essential for you to understand the nature of the application you're developing, and the requirements of the end-users for whom you're developing it.

You can start DBModeler (located in **/NextDeveloper/Apps**) from the workspace as you would any other application, by double-clicking its icon in the workspace. When it starts up, only the main menu is visible. Once DBModeler is running, you can create a new model or open an existing model as described below.

Creating, Opening, and Saving Models

Model-management commands are located in DBModeler's Model menu. To create, open, save, or close a model, use the New, Open, Save, and Close commands on this menu.

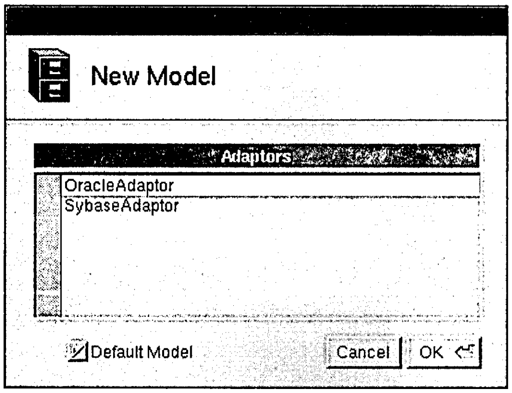


When you work in DBModeler, you'll occasionally see an authentication panel identical to or similar to the one shown here. When you see this panel (or one like it), fill in the fields and then click the OK (or Login) button.

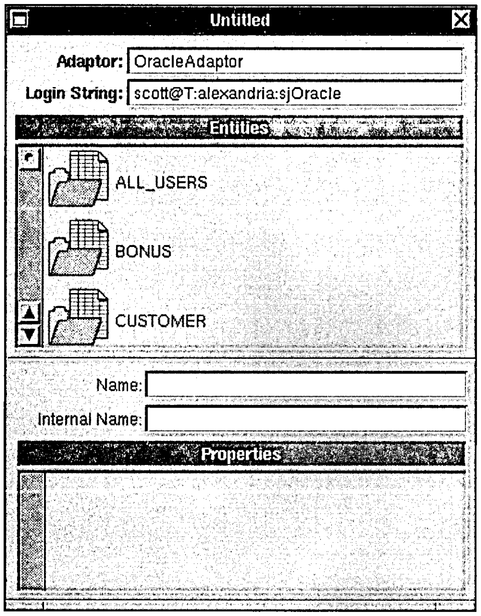
This authentication panel, which may appear at various points in working with a model, won't be discussed again. Each time the panel appears, simply provide the requested information and click Login or OK to continue.

Creating a New Model

To create a new model, choose the New command in the Model menu. The following New Model panel appears.



First, click to select the desired adaptor, and then click OK to create the new model. When the model is created, a model window similar to the one shown here will appear:



Adaptor: OracleAdaptor
Login String: scott@T:alexandria:sjOracle

For information about how to access and modify the new model through its model window, see the section "Working with Entities and Properties."

Opening an Existing Model

To open an existing model, choose the Open command in the Model menu and use the Open panel to find the model. The model must have a ".dbmodel" extension in order to appear in the Open panel. When the model opens, a model window similar to the one shown in the previous section appears.

Saving a Model

To save a new model to a file or to save changes to an existing model, choose the Save command in the Model menu. If the model hasn't been saved yet, a Save panel appears prompting you to specify a name and location for the file. The default location for models is in **~/Library/Models**. The file is saved with the standard ".dbmodel" extension, even if you don't specify it as part of the name.

Working with Entities and Properties

This section describes how to use DBModeler to work with entities and properties in a model. Entities and properties are derived from the database dictionary as follows:

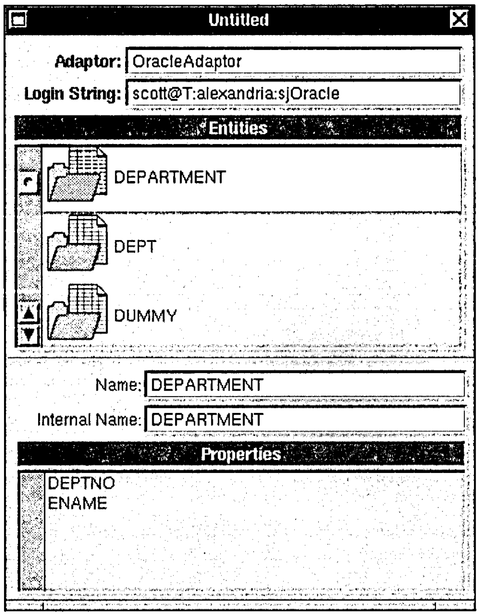
- tables in the database correspond to entities in the model
- table names in the database correspond to entity names in the model
- columns in the database correspond to attributes in the model
- column names in the database correspond to attribute names in the model

The following sections describe how you can perform the following types of operations:

- change entity or attribute names
- set the C type to be associated with a database attribute type
- designate attributes as "key"
- set read-only status for attributes
- create relationships between attributes
- designate a relationship as one-to-one or one-to-many
- designate a relationship as an inner or outer join

Entities

The center part of the model window, labeled Entities, contains a list of the entities contained in the model.



Adaptor: OracleAdaptor
Login String: scott@T:alexandria:sjOracle

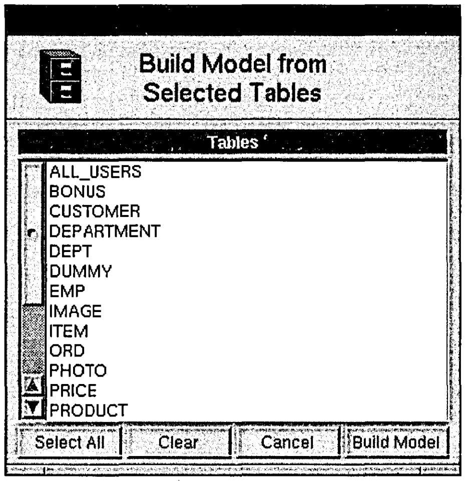
DEPARTMENT
DEPT
DUMMY

Name: DEPARTMENT
Internal Name: DEPARTMENT

DEPTNO
ENAME

Click an entity to select it. Once selected, information about the entity is displayed in the lower part of the model window. In addition to editable text fields for the name and the internal name, there's a list of the properties that belong to the entity.

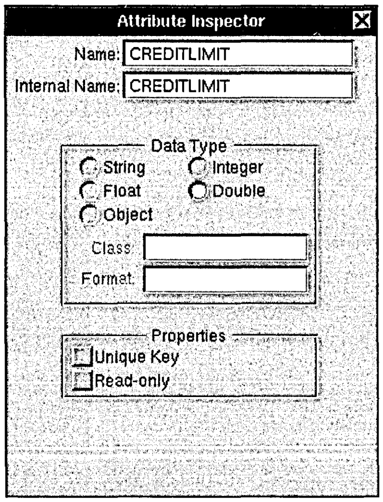
Add, hide, or delete entities by choosing the Add Entity, Hide Entity, and Delete Entity commands in the Entity menu. If you choose the Choose Entities command in the Entity menu, the following panel appears; select the tables you want to build the model from, and then click Build Model.



Properties

The lower part of the model window, labeled Properties, contains a list of the properties (attributes and relationships) contained in the selected entity.

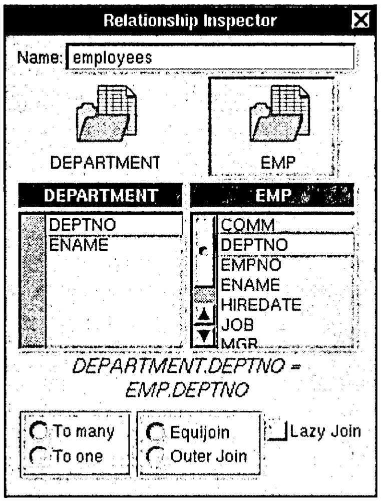
Click a property to select it. Once selected, information about the property is displayed in the Inspector panel, which appears when you choose Inspector in the main menu. The contents and appearance of the Inspector panel varies, depending on whether the selected property is an attribute or a relationship.



Name: CREDITLIMIT

Internal Name: CREDITLIMIT

If the selected property is an attribute, the Attribute Inspector appears. This panel can be used to change the name, data type, and other properties of the selected attribute.



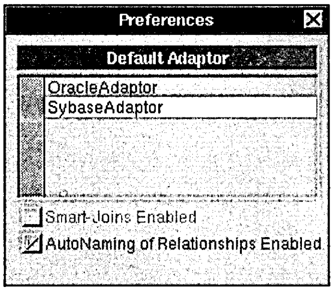
DEPARTMENT.DEPTNO = EMP.DEPTNO

If the selected property is a relationship, the Relationship Inspector appears. This panel can be used to change the name or definition of the selected relationship.

In addition to modifying properties with the Inspector panel, you can add, hide, or delete properties by choosing the Add Attribute, Add Relationship, Hide Property, and Delete Property commands in the Property menu.

Setting Preferences

Choose the Preferences command in the Info menu to bring up the Preferences panel. This panel contains a list of available adaptors—the default adaptor is highlighted. Select a different adaptor to make it the default adaptor.



☐ Smart-Joins Enabled
☑ AutoNaming of Relationships Enabled

There are also controls for determining whether smart-joins and auto-naming of relationships are enabled or disabled.

DBModeler Command Reference

This section describes the application-specific menus and commands available in DBModeler. For descriptions of standard menus and commands, see the *User's Guide*.

Commands in the Main Menu and the Model Menu

DBModeler's main menu contains the standard Info, Edit, Windows, Services, Hide, and Quit commands. The Model command displays a submenu of standard Open, New, Save, Save As, Revert to Saved, and Close commands. The Inspector command brings up the Inspector panel, which is described in the section "Properties." The Entity and Property commands display submenus that are described in the following sections.

Commands in the Entity Menu

The Entity menu provides commands for working with entities.

Command	**Description**
Add Entity	Add an entity to the current mode.
Hide Entity	Hide an entity in the current model without deleting it.
Delete Entity	Delete an entity from the current model.
Choose Entities	Choose one or more entities in the current model.

Commands in the Property Menu

The Property menu provides commands for working with the properties of an entity.

Command	**Description**
Add Attribute	Add an attribute to the selected entity.
Add Relationship	Add a relationship to the selected entity.
Hide Property	Hide a property in the selected entity, without deleting it.
Delete Property	Delete a property from the selected entity.

8 The MallocDebug Application

8 The MallocDebug Application

The Malloc Debugger measures the dynamic memory usage of applications. You can use it to measure all allocated memory in an application, or to measure the memory allocated since a given point in time. The Malloc Debugger also provides a garbage detector that you can use to detect memory leaks.

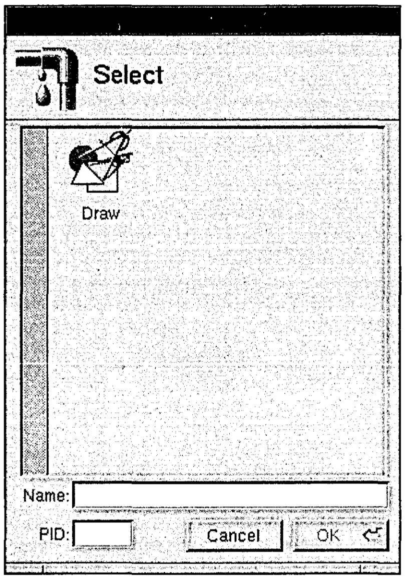
The Malloc Debugger actually consists of two components:

- A library containing a version of **malloc** that gathers statistics on memory use
- The MallocDebug application, which you use to examine those statistics

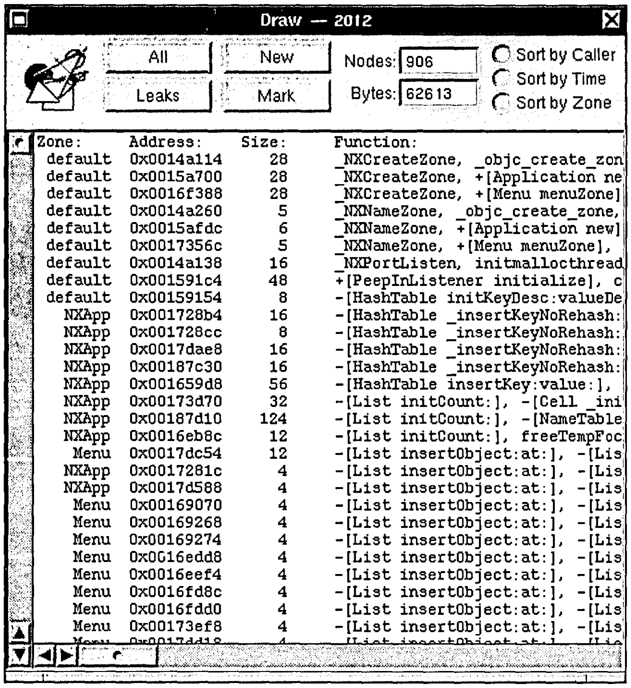
Preparing Your Application

Before using the Malloc Debugger, you must first link your application with a library containing a special version of **malloc** that can communicate with the MallocDebug application. To do this, link with the library **/usr/lib/libMallocDebug.a** using the linker option **-lMallocDebug**. The **-lMallocDebug** option must be placed before the **-lsys_s** option to ensure that **malloc** is overridden properly. If your application is built with Interface Builder, you can simply add **/usr/lib/libMallocDebug.a** to the “Other libs” section of the Project Inspector.

Using MallocDebug



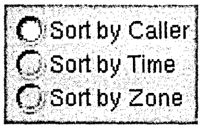
To use the Malloc Debugger, you must first select an application to monitor. Choose the Open command in the Application menu to bring up the Select panel. Only *currently running* applications that have been configured for use with MallocDebug appear in the panel. Once you select an application by double-clicking its icon, MallocDebug opens an application panel for the selected application.



```
Nodes: 906
Bytes: 62613

Zone:     Address:    Size:    Function:
 default  0x0014a114    28     _NXCreateZone, _objc_create_zon
 default  0x0015a700    28     _NXCreateZone, +[Application ne
 default  0x0016f388    28     _NXCreateZone, +[Menu menuZone]
 default  0x0014a260     5     _NXNameZone, _objc_create_zone,
 default  0x0015afdc     6     _NXNameZone, +[Application new]
 default  0x0017356c     5     _NXNameZone, +[Menu menuZone],
 default  0x0014a138    16     _NXPortListen, initmallocthread
 default  0x001591c4    48     +[PeepInListener initialize], c
 default  0x00159154     8     -[HashTable initKeyDesc:valueDe
   NXApp  0x001728b4    16     -[HashTable _insertKeyNoRehash:
   NXApp  0x001728cc     8     -[HashTable _insertKeyNoRehash:
   NXApp  0x0017dae8    16     -[HashTable _insertKeyNoRehash:
   NXApp  0x00187c30    16     -[HashTable _insertKeyNoRehash:
   NXApp  0x001659d8    56     -[HashTable insertKey:value:],
   NXApp  0x00173d70    32     -[List initCount:], -[Cell _ini
   NXApp  0x00187d10   124     -[List initCount:], -[NameTable
   NXApp  0x0016eb8c    12     -[List initCount:], freeTempFoc
    Menu  0x0017dc54    12     -[List insertObject:at:], -[Lis
   NXApp  0x0017281c     4     -[List insertObject:at:], -[Lis
   NXApp  0x0017d588     4     -[List insertObject:at:], -[Lis
    Menu  0x00169070     4     -[List insertObject:at:], -[Lis
    Menu  0x00169268     4     -[List insertObject:at:], -[Lis
    Menu  0x00169274     4     -[List insertObject:at:], -[Lis
    Menu  0x0016edd8     4     -[List insertObject:at:], -[Lis
    Menu  0x0016eef4     4     -[List insertObject:at:], -[Lis
    Menu  0x0016fd8c     4     -[List insertObject:at:], -[Lis
    Menu  0x0016fdd0     4     -[List insertObject:at:], -[Lis
    Menu  0x00173ef8     4     -[List insertObject:at:], -[Lis
```

The name of the application and the process number appear in the title bar of the application panel. Initially, the panel is empty.

Click the All button to display a list of all currently allocated nodes in your application. These nodes have been allocated by one of the standard C allocation functions (**malloc**, **realloc**, **calloc**, or **valloc**) or a NeXTSTEP zone allocation function (**NXZoneMalloc**, **NXZoneRealloc**, **NXZoneCalloc**). As shown in above, each row displays the zone in which the node was allocated, the address and the size of the node, and the function or method that allocated the node.



You can sort the nodes by caller, by time of allocation, or by zone.

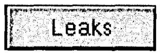
Identifying Damaged Nodes

MallocDebug detects nodes that have been written to incorrectly. If your application has written past the end of a node, a right arrow (>) appears by the node. Similarly, if your application has written before the start of a node, a left arrow (<) appears by the node. Many of these errors are caused by giving **malloc**() the result of **strlen**(s) as the argument for a string instead of **strlen**(s) **+ 1**.

Note: Damaged nodes are always listed first, regardless of the sorting mode.

Finding Memory Leaks

MallocDebug contains a conservative garbage detector, which is useful in finding memory leaks.



When you click the Leaks button, MallocDebug searches through your program's memory for pointers to each node. Any node that can't be referenced is displayed as a memory leak.

Since the garbage detector doesn't know which words in memory are pointers, it's possible that an integer has the same value as a pointer to a given node. In this case, the node doesn't show up as a leak even though it really is (this is why the garbage detector is called conservative). In practice, this problem is very rare.

Note: The garbage detector only searches for references to the beginning of each node. If your program doesn't retain a pointer to the start of a node, but instead retains a pointer into the middle of it, that node will show up as a leak even though it really isn't one.

Measuring Memory Usage

You can use MallocDebug to determine the memory usage of a given portion of your program.

To begin measuring, click the Mark button.

After exercising a portion of your program, click the New button to see the nodes allocated since the mark. MallocDebug always shows you the nodes that are still currently allocated, so you will see only those nodes allocated since the mark that haven't been freed.

MallocDebug Command Reference

This section describes the application-specific menus and commands available in MallocDebug. For descriptions of standard menus and commands, see the *User's Guide.*

Commands in the Main Menu

MallocDebug's main menu contains the standard Info, Edit, Windows, Services, Hide, and Quit commands. The Application command displays the Application menu described in the following section.

Commands in the Application Menu

The Application menu contains the following commands for opening and closing applications:

Command	Effect
Open	Displays the Select panel, from which you select and open a running application. See "Using MallocDebug" for a description of this panel.
Close	Closes the Application panel for the selected application. The debugged application remains running.

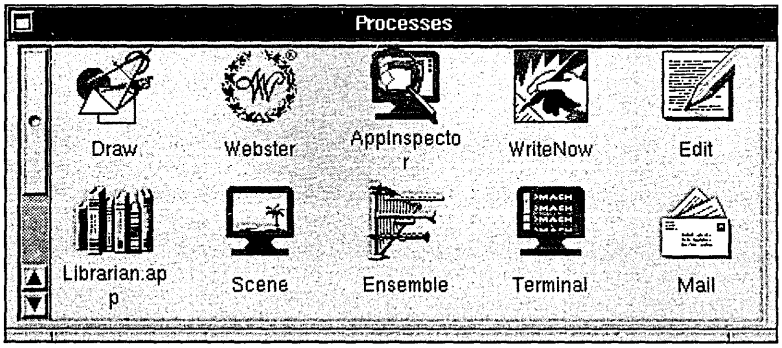
9

The Process Monitor Application

9 The Process Monitor Application

Process Monitor can be used to examine any running process. Process Monitor lets you pause or kill a process, and provides several types of information about a running process or application, including: Mach memory usage, Display PostScript®, and the run-time environment.

Selecting a Process: The Processes Panel

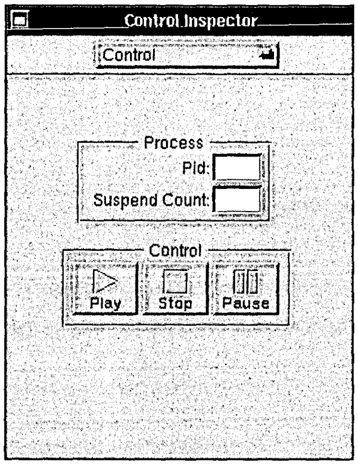


When Process Monitor starts up, the Processes panel appears. This panel contains an icon for each of the applications running on the machine. You can also see processes that aren't associated with applications by choosing the Show Non Apps command in the Processes menu.

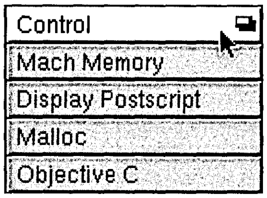
You can select any process shown in the Processes panel by clicking its icon. Once you select a process, an inspector panel appears that lets you see various types of information about the process.

You can update the contents of the Processes panel to include any processes that have been started since the panel was displayed by choosing the Update command in the Processes menu.

Inspecting a Process: The Inspector Panel

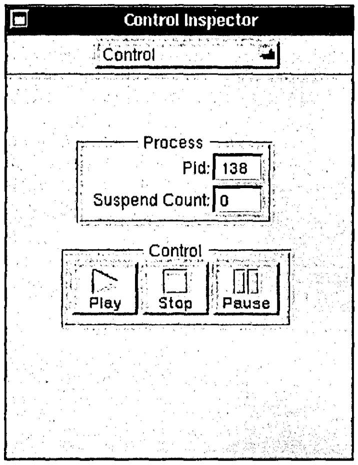


The inspector panel is actually a generic name for five different Inspectors that are available.



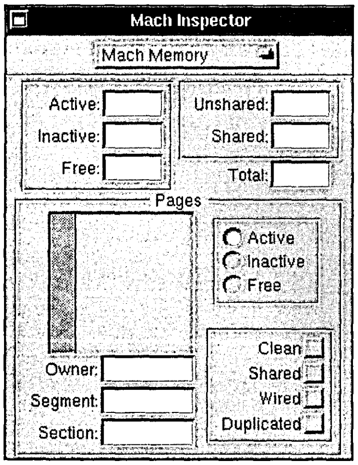
Press the button at the top of the inspector panel and drag to choose the desired Inspector. These Inspectors are described in the following sections.

The Control Inspector



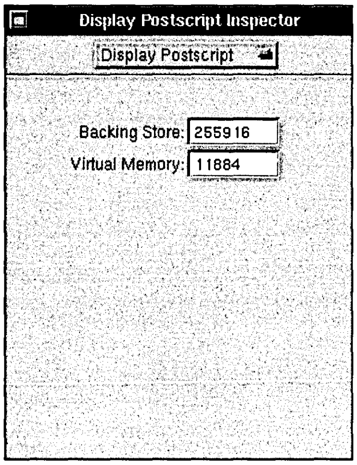
The Control Inspector displays the process ID of the selected process and indicates whether the process is suspended (paused) or running. You can use the Pause and Play buttons to pause and resume the process. Click the Stop button if you want to kill the selected process.

The Mach Inspector



The Mach Inspector displays information about the Mach memory usage of the selected process. Whenever the Mach Inspector is active, the Start Monitor command on the Monitor menu becomes enabled. The Start Monitor command opens the Mach Monitor panel, which can be used to provide a dynamic record of Mach memory usage. For more information, see "Monitoring Memory Usage: The Mach Monitor."

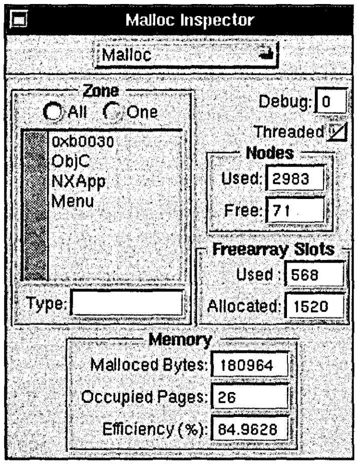
The Display PostScript Inspector



Backing Store: 255916
Virtual Memory: 11884

The Display PostScript Inspector displays information about the amount of backing store and virtual memory currently used by the selected process.

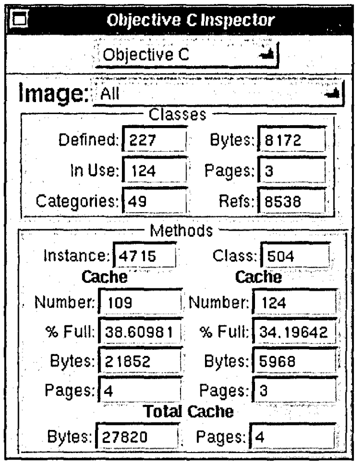
The Malloc Inspector



Type:
Debug: 0
Threaded ✓
Used: 2983
Free: 71
Used : 568
Allocated: 1520
Malloced Bytes: 180964
Occupied Pages: 26
Efficiency (%): 84.9628

The Malloc Inspector displays information about the dynamic memory usage and memory allocation efficiency of the selected process.

The Objective C Inspector



Classes

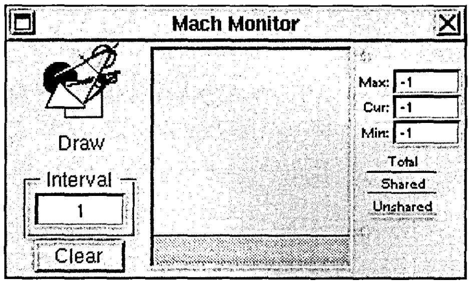
Defined:	227	Bytes:	8172
In Use:	124	Pages:	3
Categories:	49	Refs:	8538

Methods

Instance:	4715	Class:	504
Cache		Cache	
Number:	109	Number:	124
% Full:	38.60981	% Full:	34.19642
Bytes:	21852	Bytes:	5968
Pages:	4	Pages:	3
Total Cache			
Bytes:	27820	Pages:	4

The Objective C Inspector displays information about the run-time system characteristics of the selected process.

Monitoring Memory Usage: The Mach Monitor



The Mach Monitor panel appears when you choose the Monitor menu's Start Monitor command (this command is enabled only when the Mach Inspector or the Display PostScript Inspector is open). The Mach Monitor provides a running record of information about the memory usage of the monitored application or process.

To clear the contents of the Mach Monitor display, choose the Clear Monitors command from the Monitor menu.

Process Monitor Command Reference

This section describes the application-specific menus and commands available in Process Monitor. For descriptions of standard menus and commands, see the *User's Guide*.

Commands in the Main Menu

Process Monitor's main menu contains the standard Info, Edit, Windows, Print, Hide, and Quit commands. The Process and Monitor commands display submenus that are described in the following sections.

Commands in the Processes Menu

The Processes menu contains the following commands for interacting with the Processes panel:

Command	**Effect**
Update	Updates the contents of the Processes panel to include any processes that have been started since the panel was displayed. See "Selecting a Process: The Processes Panel" for a description of this panel.
Show Non Apps, Hide Non Apps	Show Non Apps causes the Processes panel to show all processes, not just those processes associated with applications. Hide Non Apps causes the panel to show just application processes.

Commands in the Monitor Menu

The Monitor menu contains the following commands for displaying and using the Mach Monitor panel:

Command	Effect
Start Monitor	Displays the Mach Monitor panel, which monitors the memory usage of the monitored application (see "Monitoring Memory Usage: The Mach Monitor"). This command is enabled only when the Mach Inspector or the Display Postscript Inspector is open.
Clear Monitors	Clears the contents of the Mach Monitor panels.

10 *The PostScript Previewers: Yap and pft*

10 *The PostScript Previewers: Yap and pft*

Yap is an interactive PostScript previewer for developers who want to write and test PostScript code. Yap lets you enter, edit, and execute PostScript code on the fly and allows you to read and write text files so the code can be used elsewhere. Yap is intended for experimenting with short, hand-created segments of PostScript. It's not useful for previewing page-oriented documents, because it ignores all Encapsulated PostScript comments (such as **%%BoundingBox** and **%%Page**). For viewing page-oriented documents, use the Preview application located in **/NextApps**.

The chapter also contains information about a related program, **pft**, which you can use if you need to communicate with the PostScript Window Server. **pft** is a command-line utility that runs in a Terminal window, so for general-purpose PostScript editing and viewing it's easier to use Yap.

Using Yap

Yap is straightforward to use. Choose the Open or New command in the Document menu to open a document window. Select the Execute command in the Document menu to execute the PostScript code that's in the main window.

The result is displayed in the Output window, and PostScript errors are reported in the title bar of the Output window. If there are no errors in the execution of your code, the execution time is reported in the title bar instead.

There's only one Output window. Its PostScript rendering area can be resized using the Preferences panel (choose the Preferences command in the Info menu). The Preferences panel also contains options for showing and clearing the PostScript cache.

If you change the font in a Yap window, that font will be used in Yap windows created after that as well. The font will also be written to your defaults database and be used the next time you launch Yap.

Yap can paste PostScript from the pasteboard; this is useful when debugging programs that write PostScript on the pasteboard. The Paste menu command first checks the pasteboard for PostScript data, then for text data.

You can find some sample PostScript programs in the following directory:

```
/NextDeveloper/Examples/PostScript
```

The following directory contains the source code for Yap:

```
/NextDeveloper/Examples/Yap
```

Feel free to modify the source code and create your own custom version of the application.

Yap Command Reference

This section describes the application-specific menus and commands available in Yap. For descriptions of standard menus and commands, see the *User's Guide*.

Commands in the Main Menu

Yap's main menu contains the standard Info, Edit, Windows, Print, Services, Hide, and Quit commands. The Format menu contains the Font command for bringing up the Font menu, and the Page Layout command.

Commands in the Document Menu

The Document menu provides the standard Open, New, Save, and Save As commands for working with PostScript document windows, plus the Execute command described here.

Command	**Description**
Execute	Executes the PostScript code contained in the main window and displays the results in the Output window.

The NeXTSTEP Window Server Interface: pft

pft is a simple shell-based utility for communicating with the NeXTSTEP Window Server. You start up the **pft** program by typing the program name in a shell window. **pft** first forms a connection to the Window Server. **pft** then sends the Window Server PostScript code that you type in the shell window, and prints out data received from the Window Server. (**pft** displays both error messages and values returned by the Window Server on the standard output, in the same window where you type.) Use Control-D to quit **pft**.

The following command-line options are available:

-NXHost *hostname*	Directs **pft** to connect to the Window Server running on the machine *hostname*. If this option isn't used, the local Window Server is assumed.
-f *file*	Causes the contents of *file* to be sent to the Window Server before user input is accepted.
-s	Causes **pft** to exit after a file specified with **-f** is sent to the Window Server.
-NXPSName *string*	Sets the string that **pft** uses to find the Window Server that it will connect to. This should be the name that the Window Server used to register itself with **nmserver**, the Network Message Server. If this option isn't used, the default Window Server name is assumed.

pft sends one line of PostScript to the Window Server at a time, and each line is interpreted by the Window Server immediately after you press Return.

Starting the pft Program

To run the **pft** program, enter its name in a shell window:

```
pft
```

When **pft** responds with "Connection to PostScript established," it's ready to accept PostScript code. If you're running **pft** in a Terminal window, you can cut and paste PostScript code from another application.

When you're finished, quit by typing Control-D (or Control-C) in the shell window that **pft** is running in.

Executing PostScript Code from a File

To execute PostScript commands that are contained in a file, you can start **pft** using the **-f** option:

pft -f *file*

The *file* argument must be an absolute pathname (that is, starting with either **/** or **~**), as shown in these two examples:

```
pft -f /me/myProgram.ps
pft -f ~/myProgram.ps
```

Alternatively, once you've started running **pft** the contents of a PostScript file can be executed using the PostScript **run** operator:

(*file*) **run**

In this case, the file name must be an absolute pathname that doesn't start with ~:

```
(/me/myProgram.ps) run
```

Setting Up a Window

The first thing you'll probably want to do in **pft**, once it has established a connection to the Window Server, is set up a window to draw in. There are two ways to do this:

- Obtain the window number of a window the Server has already set up for some other application (usually one you are using **pft** to debug), and do your drawing in that window.
- Set up a new window using the PostScript **window** operator.

To create a window with the **window** operator, pass it arguments for its origin, size, and type:

x y width height type **window** *window*

where *type* is one of **Retained, Nonretained**, or **Buffered**:

Retained	(0)
Nonretained	(1)
Buffered	(2)

The **window** operator returns a unique ID number for the window, and places this number on the operand stack. You'll need this number in order to refer to the window; for ease of reference you can assign the returned window number to a variable, as follows:

```
/myWindow
100 100 500 500 Buffered window
def
```

The new window isn't in the screen list yet, and therefore doesn't appear on the screen and doesn't receive user events. You can add the window to the screen list with the **orderwindow** operator:

place otherwindow window **orderwindow** –

The location of the window in the screen list is specified by *place*, which can be one of **Below, Out**, or **Above**:

Below	(–1)
Out	(0)
Above	(1)

otherwindow should be another window number, or 0 if you want to place the new window above or below all windows currently in the window list.

Once the window is in the screen list it appears on the screen, but before you can draw in the window you need to use the **windowdeviceround** operator to make the window the current window:

window **windowdeviceround** –

Once the window is the current window, the results of any drawing code you enter will be displayed:

```
newpath
20 20 moveto
40 40 lineto
stroke
flushgraphics  % necessary if window is buffered
```

Flushing the Server's Output Buffer

The connection between **pft** and the Window Server is buffered in both directions. **pft** flushes its input buffer, so none of the PostScript you send to the Window Server is ever caught in the buffer. However, you must flush the Window Server's output buffer yourself using the PostScript **flush** operator.

Here's a one-line example showing how to create a 500-pixel by 500-pixel window whose lower left corner is at the lower left corner of the screen. This example removes the window number from the stack and flushes the Window Server's output buffer:

```
0 0 500 500 Buffered window = flush
```

Summary Example

In summary, this simple series of PostScript commands demonstrates how to create a window, draw in the window, and then remove the window:

```
/myWindow                      % Create a variable called myWindow
100 100 50 50 Buffered window % Create a window, and assign the returned
def                            % window number to the myWindow variable

Above 0 myWindow orderwindow   % Order myWindow at front of screen list
myWindow windowdeviceround     % Make myWindow the current window

newpath                        % Now draw something to myWindow
25 25 15 0 360 arc
fill
flushgraphics                   % Flushing is required for buffered windows

myWindow termwindow            % Mark myWindow for destruction
nulldevice                     % Remove references to myWindow
```

11 *The GNU C Compiler*

11 *The GNU C Compiler*

The C compiler used on NeXTSTEP computers is GNU CC, an ANSI-standard C compiler produced by the Free Software Foundation. This compiler has been modified and extended as a compiler for the Objective C language by NeXT Computer, Inc. for use on NeXTSTEP computers. This chapter describes how to compile a C program using the GNU compiler.

This chapter is a modified version of documentation provided by the Free Software Foundation; see the section Legal Considerations at the end of the chapter for important related information.

The following sections describe command options available when compiling a C program with GNU CC, incompatibilities between GNU CC and non-ANSI versions of C, GNU extensions to the C language, and implementation-specific details related to using C on a NeXT computer.

GNU CC Command Options

When you invoke GNU CC with the **cc** command, it normally performs the following operations in the order shown here:

- Preprocessing (**cpp**)
- Compilation (**cc1**)
- Assembly (**as**)
- Linking (**ld**)

Some options described below allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these aren't documented here, since you rarely need to use any of them.

The GNU C compiler uses a command syntax much like the UNIX C compiler. The **gcc** program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: **-dr** is very different from **-d -r**.

Many options have long names starting with **-f** (**-fforce-mem**, **-fstrength-reduce**, and so on). Most of these have both positive and negative forms; the negative form of **-ffoo** would be **-fno-foo**. This manual documents only one of these two forms—whichever one *isn't* the default.

Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; the fourth stage—linking—combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done. For example, a ".c" file is C source code which must be preprocessed, a ".i" file is C source code which shouldn't be preprocessed, a ".cc" file is C++ source code which must be preprocessed, a ".s" file is assembler code, and an unrecognized file name is considered an object file and is fed straight into linking.

You can specify the input language explicitly with the **-x** option:

-x *language*

Specify that the following input files are in the language *language*. This option applies to all following input files until the next **-x** option. Possible values of language are **c**, **objective-c**, **c-header**, **c++**, **cpp-output**, **assembler**, and **assembler-with-cpp**.

-x none

Turn off any specification of a language, so that subsequent files are handled according to their file-name suffixes (as they are if **-x** has not been used at all).

The point at which the compilation process stops is controlled by various options:

-c Compile or assemble the source files, but don't link. The linking stage simply isn't done. The ultimate output is in the form of an object file for each source file. By default, the object file name for a source file is made by replacing the suffix ".c", ".i", ".s", and so on, with ".o". Unrecognized input files, not requiring compilation or assembly, are ignored.

-S Stop after the stage of compilation proper; don't assemble. The output is in the form of an assembler code file for each non-assembler input file specified. By default, the assembler file name for a source file is made by replacing the suffix ".c", ".i", etc., with ".s". Input files that don't require compilation are ignored.

-E Stop after the preprocessing stage; don't run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. Input files which don't require preprocessing are ignored.

-o *file* Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. (Since only one output file can be specified, it doesn't make sense to use **-o** when compiling more than one input file, unless you are producing an executable file as output.)

If **-o** isn't specified, the default is to put an executable file in **a.out**, the object file for *source.suffix* in *source*.**o**, its assembler file in *source*.**s**, and all preprocessed C source on the standard output.

-v Print (to standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-vpath Print (to standard error output) all attempts at finding files, tracing how the **-B**, **-b**, and **-V** options interact. Also, print the commands executed to run the stages of compilation and version numbers, like the **-v** option.

-vspec Print (to standard error output) all spec's processed by the **do_spec_1()** function in **gcc.c**. Also, print the commands executed to run the stages of compilation and version numbers, like the **-v** option.

-pipe Use pipes rather than temporary files for communication between the various stages of compilation.

-B*path* Compiler driver program tries *path* (which must end in /) as the directory prefix for each program it tries to run. These programs are **cpp**, **cc1**, **as**, and **ld**.

For each subprogram to be run, the compiler driver first tries the **-B** prefix, if any. If that name isn't found, or if **-B** wasn't specified, the driver tries two standard prefixes, **/bin/** and **/lib/**. If neither of those results in a file name that's found, the unmodified program name is searched for using the directories specified in your PATH environment variable.

Specifying a Dialect of the C Language

The following options control the dialect of C that the compiler accepts:

-ansi Support all ANSI C programs. This turns off certain features of GNU C that are incompatible with ANSI C, such as the **asm**, **inline** and **typeof** keywords, and predefined macros such as **unix** and **vax** that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature.

The alternate keywords **__asm__**, **__extension__**, **__inline__**, and **__typeof__** continue to work despite **-ansi**. You wouldn't want to use them in an ANSI C program, of course, but it useful to put them in header files that might be included in compilations done with **-ansi**. Alternate predefined macros such as **__unix__** and **__vax__** are also available, with or without **-ansi**.

The **-ansi** option doesn't cause non-ANSI C programs to be rejected gratuitously. For that, **-pedantic** is required in addition to **-ansi**. See the section Requesting or Suppressing Warnings for more information.

The macro **__STRICT_ANSI__** is predefined when the **-ansi** option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

-ObjC Compile a source file that contains Objective C language code (the file can have either a ".c" or an ".m" extension).

-bsd Enforce strict BSD semantics. When the **-bsd** option is used, the macro **__STRICT_BSD__** is predefined in the preprocessor. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros.

-trigraphs Support ANSI C trigraphs. The **-ansi** option implies **-trigraphs.**

-traditional

Attempt to support some aspects of traditional C compilers. Specifically:

- All extern declarations take effect globally even if they are written inside a function definition. This includes implicit declarations of functions.
- The keywords **typeof, inline, signed, const**, and **volatile** aren't recognized. (You can still use the alternative keywords such as **__typeof__**, **__inline__**, and so on.)
- Comparisons between pointers and integers are always allowed.
- Integer types **unsigned short** and **unsigned char** promote to **unsigned int**.
- Out-of-range floating point literals aren't an error.
- String "constants" aren't necessarily constant; they are stored in writable space, and identical-looking constants are allocated separately. (This is the same as the effect of **-fwritable-strings**.)
- All automatic variables not declared **register** are preserved by **longjmp()**. Ordinarily, GNU C follows ANSI C: automatic variables not declared **volatile** may be clobbered.
- In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quotation marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.
- The predefined macro **__STDC__** isn't defined when you use **-traditional**, but **__GNUC__** is (since the GNU extensions which **__GNUC__** indicates aren't affected by **-traditional**). If you need to write header files that work differently depending on whether **-traditional** is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.

-gnu-cpp Use the GNU C preprocessor instead of the NeXTSTEP C preprocessor.

-fno-asm Don't recognize **asm**, **inline** or **typeof** as a keyword. These words may then be used as identifiers. You can use **__asm__**, **__inline__**, and **__typeof__** instead. **-ansi** implies **-fno-asm**.

-fcond-mismatch

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

-funsigned-char

Let the type **char** be the unsigned, like **unsigned char**. (Note that each type of computer has a default for what **char** should be—it's either like **unsigned char** by default or like **signed char** by default. The type **char** is always a distinct type from either **signed char** or **unsigned char**, even though its behavior is always just like one of those two.)

Ideally, a portable program should always use **signed char** or **unsigned char** when it depends on the signedness of an object. But many programs have been written to use plain **char** and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

-fsigned-char

Let the type **char** be signed, like **signed char**. Note that this is equivalent to **-fno-unsigned-char**, which is the negative form of **-funsigned-char**. Likewise, **-fno-signed-char** is equivalent to **-funsigned-char**.

-fsigned-bitfields, -funsigned-bitfields, -fno-signed-bitfields, -fno-unsigned-bitfields

Similar to the above flags, these options control whether a bitfield is signed or unsigned, when the declaration doesn't use either signed or unsigned. By default, such a bitfield is signed, because this is consistent: the basic integer types such as **int** are signed types. However, when **-traditional** is used, bitfields are all unsigned no matter what.

-fwritable-strings

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. **-traditional** also has this effect. (Note that writing into string constants is a bad idea; "constants" should be constant.)

Requesting or Suppressing Warnings

Warnings are diagnostic messages that report constructions that aren't inherently erroneous, but which are risky or suggest there may have been an error.

These options control the amount and kinds of warnings produced by GNU CC:

-w Inhibit all warning messages.

-pedantic Issue all the warnings demanded by strict ANSI C; reject all programs that use forbidden extensions.

Valid ANSI C programs should compile properly with or without this option (though a rare few will require **-ansi**). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected. There's no reason to use this option; it exists only to satisfy pedants.

-pedantic doesn't cause warning messages for use of the alternate keywords whose names begin and end with __. Pedantic warnings are also disabled in the expression that follows **__extension__**. However, only system header files should use these escape routes; application programs should avoid them.

-pedantic-errors
Like **-pedantic**, except that errors are produced rather than warnings.

-W Print extra warning messages for these events:

- A nonvolatile automatic variable might be changed by a call to **longjmp()**. These warnings as well are possible only in optimizing compilation.

- The compiler sees only the calls to **setjmp()**. It cannot know where **longjmp()** will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there's in fact no problem because **longjmp()** cannot in fact be called at the place which would cause a problem.

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
    if (a > 0)
    return a;
}
```

  Spurious warnings can occur because GNU CC doesn't realize that certain functions (including **abort()** and **longjmp()**) will never return.

- An expression-statement contains no side effects.
- An unsigned value is compared against zero with **>** or **<=**.

-Wimplicit
Warn whenever a function or parameter is implicitly declared.

-Wreturn-type
Warn whenever a function is defined with a return-type that defaults to **int.** Also warn about any return statement with no return value in a function whose return type isn't **void**.

-Wunused Warn whenever a local variable is unused aside from its declaration, whenever a function is declared **static** but never defined, and whenever a statement computes a result that is explicitly not used.

-Wswitch Warn whenever a switch statement has an index of enumeral type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) Case labels outside the enumeration range also provoke warnings when this option is used.

-Wcomment
Warn whenever a comment-start sequence /* appears in a comment.

-Wtrigraphs
Warn if any trigraphs are encountered (assuming they are enabled).

-Wformat Check calls to **printf()**, **scanf()**, and so on, to make sure that the arguments supplied have types appropriate to the format string specified.

-Wuninitialized
Warn if an automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify **-O**, you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they don't occur for a variable that is declared **volatile**, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they don't occur for structures, unions, or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC isn't smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here's one example of how this can happen:

```
{
  int x;
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of **y** is always 1, 2 or 3, then **x** is always initialized, but GNU CC doesn't know this. Here's another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  . . .
  if (change_y) y = save_y;
}
```

This has no bug because **save_y** is used only if it's set.

Some spurious warnings can be avoided if you declare as volatile all the functions you use that never return.

-Wall All of the above **-W** options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining **-W** options aren't implied by **-Wall** because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

-Wtraditional

Warn about certain constructs that behave differently in traditional and ANSI C:

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.
- A function declared external in one block and then used after the end of the block.
- A switch statement has an operand of type **long**.

-Wshadow

Warn whenever a local variable shadows another local variable.

-Wid-clash-*len*

Warn whenever two distinct identifiers match in the first *len* characters. This may help you prepare a program that will compile with certain obsolete compilers.

-Wpointer-arith

Warn about anything that depends on the "size of" a function type or of **void**. GNU C assigns these types a size of 1, for convenience in calculations with **void** * pointers and pointers to functions.

-Wcast-qual

Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a **const char** * is cast to an ordinary **char** *.

-Wcast-align

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a **char** * is cast to an **int** * on machines where integers can only be accessed at two-byte or four-byte boundaries.

-Wwrite-strings

Give string constants the type **const char**[*length*] so that copying the address of one into a non-**const char** * pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using **const** in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make **-Wall** request these warnings.

-Wconversion

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

Preparing Your Program for Debugging

GNU CC has various special options that are used for debugging either your program or GCC:

-g

Produce debugging information for use with GDB.

Unlike most other C compilers, GNU CC allows you to use **-g** with **-O**. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

-pg

Generate extra code to write profile information suitable for the analysis program **gprof**.

-d*letters* Make debugging dumps during compilation at times specified by *letters*. This is used for debugging the compiler. The file names for most of the dumps are made by appending a word to the source file name (e.g., **foo.c.rtl** or **foo.c.jump**). Here are the possible letters:

y	Dump debugging information during parsing, to standard error.
r	Dump after RTL generation, to *file*.**rtl**.
x	Just generate RTL for a function instead of compiling it. Usually used with **r**.
j	Dump after first jump optimization, to *file*.**jump**.
s	Dump after CSE (including the jump optimization that sometimes follows CSE), to *file*.**cse**.
L	Dump after loop optimization, to *file*.**loop**.
t	Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to *file*.**cse2**.
f	Dump after flow analysis, to *file*.**flow**.
c	Dump after instruction combination, to *file*.**combine**.
S	Dump after the first instruction scheduling pass, to *file*.**sched**.
l	Dump after local register allocation, to *file*.**lreg**.
g	Dump after global register allocation, to *file*.**greg**.
R	Dump after the second instruction scheduling pass, to *file*.**sched2**.
J	Dump after last jump optimization, to *file*.**jump2**.
d	Dump after delayed branch scheduling, to *file*.**dbr**.
k	Dump after conversion from registers to stack, to *file*.**stack**.
m	Print statistics on memory usage to standard error, at the end of the run.
p	Annotate the assembler output with a comment indicating which pattern and alternative was used.

-fpretend-float

When running a cross-compiler, pretend that the target machine uses the same floating-point format as the host machine. This causes incorrect output of the actual floating constants, but the actual instruction sequence will probably be the same as GNU CC would make when running on the target machine.

-save-temps

Store the usual "temporary" intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling **foo.c** with **-c -save-temps** would produce files **foo.cpp** and **foo.s**, as well as **foo.o**.

Controlling Optimization

These options control various sorts of optimizations:

-O

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without **-O**, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. Also, only variables declared **register** are allocated in registers.

With **-O**, the compiler tries to reduce code size and execution time; also, **-fthread-jumps** and **-fdelayed-branch** are turned on.

-O2

Highly optimize. All supported optimizations that don't involve a space-speed tradeoff are performed. As compared to **-O**, this option will increase both compilation time and the performance of the generated code. All **-fflag** options that control optimization are turned on when **-O2** is specified.

Options of the form **-f***flag* specify machine-independent flags. Most flags have both positive and negative forms; the negative form of **-ffoo** would be **-fno-foo**. In the table below, only one of the forms is listed—the one which *isn't* the default. You can figure out the other form by either removing **no-** or adding it.

-ffloat-store

Don't store floating point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a double is supposed to have.

For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use **-ffloat-store** for such programs.

-fno-defer-pop

Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

-fforce-mem

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they aren't common subexpressions, instruction combination should eliminate the separate register-load.

-fforce-addr

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as **-fforce-mem** may.

-fomit-frame-pointer

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. It also makes debugging impossible on most machines.

-finline Pay attention to the **inline** keyword. Normally the negation of this option **-fno-inline** is used to keep the compiler from expanding any functions inline. However, the opposite effect may be desirable when compiling with **-g**, since **-g** normally turns off all inline function expansion.

-finline-functions

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared **static**, then the function is normally not output as assembler code in its own right.

-fcaller-saves

Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

-fkeep-inline-functions

Even if all calls to a given function are integrated and the function is declared static, nevertheless output a separate run-time callable version of the function.

-fno-function-cse

Don't put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option isn't used.

The following options control specific optimizations. The **-O2** option turns on all of these optimization except **-funroll-loops** and **-funroll-all-loops**. The **-O** option usually turns on the **-fthread-jumps** and **-fdelayed-branch** options, but specific machines may change the default optimizations.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

-fstrength-reduce

Perform the optimizations of loop strength reduction and elimination of iteration variables.

-fthread-jumps

Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

-funroll-loops

Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time.

-funroll-all-loops

Perform the optimization of loop unrolling. This is done for all loops. This usually makes programs run more slowly.

-fcse-follow-jumps

In common subexpression elimination, scan through jump instructions in certain cases. This isn't as powerful as completely global CSE, but not as slow either.

-frerun-cse-after-loop

Re-run common subexpression elimination after loop optimizations has been performed.

-fexpensive-optimizations

Perform a number of minor optimizations that are relatively expensive.

-fdelayed-branch

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

-fschedule-insns

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

-fschedule-insns2

Similar to **-fschedule-insns**, but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the **-E** option, nothing is done except preprocessing. Some of these options make sense only together with **-E**, because they cause the preprocessor output to be unsuitable for actual compilation.

-i *file* Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of **-i** *file* is to make the macros defined in *file* available for use in the main input.

-nostdinc Don't search the standard system directories for header files. Only the directories you have specified with **-I** options (and the current directory, if appropriate) are searched. See the section Specifying Directories to be Searched for more information on **-I**.

Between **-nostdinc** and **-I-**, you can eliminate all directories except those specified explicitly from the search path for header files.

-E Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.

-C Tell the preprocessor not to discard comments. Used with the **-E** option.

-P Tell the preprocessor not to generate **#line** commands. Used with the **-E** option.

-M Tell the preprocessor to output a rule suitable for **make** describing the dependencies of each object file. For each source file, the preprocessor outputs one **make** rule whose target is the object file name for that source file and whose dependencies are all the files **#included** in it. This rule may be a single line or may be continued with backslash-newline if it's long. The list of rules is printed on standard output instead of the preprocessed C program.

-M implies **-E**.

-MM Like **-M** but the output mentions only the user header files included with **#include** "*file*". System header files included with **#include** <*file*> are omitted.

-MD Like **-M** but the dependency information is written to files with names made by replacing ".c" with ".d" at the end of the input file names. This is in addition to compiling the file as specified—note that **-MD** doesn't inhibit ordinary compilation the way **-M** does.

The Mach utility **md** can be used to merge the ".d" files into a single dependency file suitable for using with the **make** command.

-MMD Like **-MD** except mention only user header files, not system header files.

-H Print the name of each header file used, in addition to other normal activities.

-D*macro* Define macro *macro* with the string **1** as its definition.

-D*macro=defn*
Define macro *macro* as *defn*.

-U*macro* Undefine macro *macro*.

-trigraphs Support ANSI C trigraphs. The **-ansi** option also has this effect.

Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler isn't doing a link step.

object-file-name
A file name that doesn't end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

-c, -S, -E If any of these options is used, then the linker isn't run, and object file names shouldn't be used as arguments. See the section Controlling the Kind of Output for more information.

-nostdlib Don't use the standard system libraries and startup files when linking. Only the files you specify will be passed to the linker.

Additional linker options are described in the **ld**(1) UNIX manual page.

Specifying Directories to be Searched

These options specify directories to search for header files, for libraries and for parts of the compiler:

-I*dir*	Search directory *dir* for include files.
-I-	Any directories specified with **-I** options before the **-I-** option are searched only for the case of **#include** "*file*"; they aren't searched for **#include** <*file*>.

If additional directories are specified with **-I** options after the **-I-**, these directories are searched for all **#include** directives. (Ordinarily all **-I** directories are used this way.)

In addition, the **-I-** option inhibits the use of the current directory (where the current input file came from) as the first search directory for **#include** "*file*". There's no way to override this effect of **-I-**. With **-I.** you can specify searching the directory which was current when the compiler was invoked. That isn't exactly the same as what the preprocessor does by default, but it's often satisfactory.

-I- doesn't inhibit the use of the standard system directories for header files. Thus, **-I-** and **-nostdinc** are independent.

-L*dir*	Add directory *dir* to the list of directories to be searched for **-l**.

Specifying Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of **-ffoo** would be **-fno-foo**. In the table below, only one of the forms is listed—the one which *isn't* the default. You can figure out the other form by either removing **no-** or adding it.

-fshort-enums

Allocate to an **enum** type only as many bytes as it needs for the declared range of possible values. Specifically, the **enum** type will be equivalent to the smallest integer type which has enough room.

-fno-common
Alocate even uninitialized global variables in the **bss** section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without **extern**) in two different compilations, you'll get an error when you link them.

-fvolatile Consider all memory references through pointers to be volatile.

-ffixed-*reg* Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the REGISTER_NAMES macro in the machine description macro file.

This flag doesn't have a negative form, because it specifies a three-way choice.

-fcall-used-*reg*
Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that don't live across a call. Functions compiled this way won't save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag doesn't have a negative form, because it specifies a three-way choice.

-fcall-saved-*reg*
Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results. A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag doesn't have a negative form, because it specifies a three-way choice.

C Programming Notes

This section contains miscellaneous notes about programming in C on a NeXTSTEP computer. It also describes some incompatibilities between GNU C and traditional non-ANSI versions of C.

String Constants and Static Strings

GNU CC normally makes string constants read-only, and if several identical string constants are used, GNU CC stores only one copy of the string.

Some C libraries incorrectly write into string constants. The best solution to this problem is to use character array variables with initialization strings instead of string constants. If this isn't possible, use the **-fwritable-strings** flag, which directs GNU CC to handle string constants the way most C compilers do.

Also note that initialized strings are normally put in the text segment by the GNU compiler, and attempts to write to them cause segmentation faults. If your program depends on being able to write initialized strings, there are two ways to get around this problem:

- Compile your program with the **-fwritable-strings** compiler option.
- Declare your string as an unbounded array of **char**s, which will force it to appear in the data segment:

```
char *non_writable = "You can't write this string";
char writable[] = "You can write this string";
```

Function Prototyping

Function prototypes are a new and important feature of the ANSI standard. You should use function prototypes in your C programs, so the compiler can generate more efficient code (because it knows what the called function is expecting). The compiler can also warn you when you pass the wrong number or wrong type of arguments to a function.

Extra care must be taken in using function prototypes. Be sure to follow these rules:

- Each function must be declared explicitly (with a prototype) before calling the function. Multiple declarations must agree exactly. Incorrect code can be generated by a call that isn't prototyped if the function itself is declared as a prototype.
- The parameter declarations for the prototyped function must be in the same form as the prototype declaration.

Here are a few points about prototyping that might cause you some trouble.

- You might think it's a bug when GNU CC reports an error for code like this:

  ```
  int foo (short);

  int foo (x)
      short x;
  { . . . }
  ```

  The error message is correct. The code is wrong because the old-style nonprototype definition passes subword integers in their promoted types. In other words, the argument is really an **int**, not a **short**. The correct prototype is this:

  ```
  int foo (int)
  ```

- You might think it's a bug when GNU CC reports an error for code like this:

  ```
  int foo (struct mumble *);

  struct mumble { . . . };

  int foo (struct mumble *x);
  { . . . }
  ```

  This code is also wrong. Because of the scope of **struct mumble**, the prototype is limited to the argument list containing it. It doesn't refer to the **struct mumble** defined with file scope immediately below—they are two unrelated types with similar names in different scopes. But in the definition of **foo**, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype don't match and you get an error. You can make the code work by simply moving the definition of **struct mumble** above the prototype.

"Suggested Reading" lists several C books that provide detailed information about the use (and abuse) of function prototypes.

Automatic Register Allocation

When you use **setjmp()** and **longjmp()**, the only automatic variables guaranteed to remain valid are those declared **volatile**. This is a consequence of automatic register allocation. If you use the **-W** option with the **-O** option, you'll get a warning when GNU CC thinks such a problem is possible. For example:

```
jmp_buf j;

foo ()
{
    int a, b;

    a = fun1 ();
    if (setjmp (j))
        return a;

    a = fun2 ();
    /* longjmp (j) may occur in fun3. */
    return a + fun3 ();
}
```

Here, **a** may or may not be restored to its first value when the **longjmp()** function is called. If **a** is allocated in a register, its first value is restored; otherwise, it keeps the last value stored in it.

Declarations of External Variables and Functions

Declarations of external variables and functions within a block apply only to the block containing the declaration (in some C compilers, such declarations affect the whole file). ANSI C states that external declarations should obey normal scoping rules. For example:

```
{
    {
        extern int a;
        a = 0;
    }
    a = 1;      /* Illegal */
}
```

You can use the **-traditional** option if you want all **extern** declarations to be treated as global.

typedef and Type Modifiers

In traditional C, you can combine **unsigned**, for example, with a **typedef** name as shown here:

```
typedef long int Int32;
unsigned Int32 i;       /* Illegal in ANSI C*/
```

In ANSI C this isn't allowed: **unsigned** and other type modifiers require an explicit **int.** Because this criterion is expressed by Bison grammar rules rather than C code, the **-traditional** flag can't alter it.

The same difficulty applies to **typedef** names used as function parameters.

GNU Extensions to the C Language

GNU C provides several language features not found in ANSI C. (The **-pedantic** option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro **__GNUC__**, which is always defined under GNU CC.

Note: You should avoid the use of these GNU C extensions to the ANSI C language, since they aren't guaranteed to be supported in future releases of NeXTSTEP.

Casts as Lvalues

In GNU C, casts are allowed as lvalues provided their operands are lvalues. This means that you can store values into them.

A cast is a valid lvalue if its operand is an lvalue. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if **a** has type **char ***, the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)(int)5)
```

An assignment-with-arithmetic operation such as **+=** applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *)(int) ((int)a + 5))
```

You cannot take the address of an lvalue cast, because the use of its address wouldn't work out coherently. Suppose that **&(int)f** were permitted, where **f** has type **float**. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

```
*&(int)f = 1;
```

This is quite different from what **(int)f = 1** would do; that would convert 1 to floating point and store it. Rather than cause this inconsistancy, we think it's better to prohibit use of **&** on a cast.

If you really do want an **int** * pointer with the address of **f**, you can simply write **(int *)&f.**

Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline
    = (struct line *) malloc (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give **contents** a length of 1, which means either you waste space or complicate the argument to **malloc**.

Arithmetic on void-Pointers and Function Pointers

In GNU C, addition and subtraction operations are supported on pointers to **void** and on pointers to functions. This is done by treating the size of a **void** or of a function as 1.

A consequence of this is that **sizeof()** is also allowed on **void** and on function types, and returns 1.

The option **-Wpointer-arith** requests a warning if these extensions are used.

Non-Constant Initializers

The elements of an aggregate initializer for an automatic variable aren't required to be constant expressions in GNU C. Here's an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  . . .
}
```

Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer.

Usually, the specified type is a structure. Assume that **struct foo** and **structure** are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here's an example of constructing a **struct foo** with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are made up of simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements aren't simple constants aren't very useful, because the constructor isn't an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a switch statement, while the latter does the same thing an ordinary C initializer would do. Here's an example of subscripting an array constructor:

```
output = ((int[]) { 2, x, 28 }) [input];
```

Constructor expressions for scalar types and union types are is also allowed, but then the constructor expression is equivalent to a cast.

Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls.

A few functions, such as **abort()** and **exit()**, cannot return. These functions should be declared **volatile**. For example,

```
extern void volatile abort ();
```

tells the compiler that it can assume that **abort()** won't return. This makes slightly better code, but more importantly it helps avoid spurious warnings of uninitialized variables. It doesn't make sense for a volatile function to return anything other than **void**.

Many functions don't examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared **const**. For example,

```
extern int const square ();
```

says that the hypothetical function **square()** is safe to call fewer times than the program says.

Note that a function that has pointer arguments and examines the data pointed to must not be declared **const**. Likewise, a function that calls a non-**const** function usually must not be **const**. It doesn't make sense for a **const** function to return **void**.

We recommend placing the keyword **const** after the function's return type. It makes no difference in the example above, but when the return type is a pointer, it's the only way to make the function itself **const**. For example,

```
const char *mincp (int);
```

says that **mincp()** returns **const char** *—a pointer to a **const** object. To declare **mincp()** as **const**, you must write this:

```
char * const mincp (int);
```

Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

Dollar signs are allowed on certain machines if you specify **-traditional.** On a few systems they are allowed by default, even if **-traditional** isn't used. But they are never allowed if you specify **-ansi.**

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$
lose (test)
```

The Character ESC in Constants

In GNU C, you can use the sequence **\e** in a string or character constant to stand for the ASCII character ESC.

Specifying Attributes of Variables

In GNU C, the keyword **__attribute__** allows you to specify special attributes of variables or structure fields. The only attributes currently defined are the **aligned** and **format** attributes.

The **aligned** attribute specifies the alignment of the variable or structure field. For example, the declaration

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable **x** on a 16-byte boundary. On a 68000, this could be used in conjunction with an **asm** expression to access the **move16** instruction, which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned int pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a **double** member that forces the union to be double-word aligned.

It isn't possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed.

The **format** attribute specifies that a function takes **printf()** or **scanf()** style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to **my_printf()** for consistency with the printf-style format string argument **my_format**.

The first parameter of the **format** attribute determines how the format string is interpreted, and should be either **printf** or **scanf**. The second parameter specifies the number of the format string argument (starting from 1). The third parameter specifies the number of the first argument which should be checked against the format string. For functions where the arguments aren't available to be checked (such as **vprintf()**), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string **(my_format)** is the second argument to **my_print()** and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The **format** attribute allows you to identify your own functions which take format strings as arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI C library functions **printf()**, **fprintf()**, **sprintf()**, **scanf()**, **fscanf()**, **sscanf()**, **vprintf()**, **vfprintf()**, and **vsprintf()** whenever such warnings are requested (using **-Wformat**), so there's no need to modify the header file **stdio.h.**

An Inline Function is As Fast As a Macro

By declaring a function inline, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

To declare a function inline, use the **inline** keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

If you are writing a header file to be included in ANSI C programs, write **__inline__** instead of **inline**.

You can also make all "simple enough" functions inline with the option **-finline-functions**. Note that certain usages in a function definition can make it unsuitable for inline substitution.

When a function is both inline and static, if all calls to the function are integrated into the caller and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC doesn't actually output assembler code for the function, unless you specify the option **-fkeep-inline-functions**. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there's a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function isn't static, the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-static inline function is always compiled on its own in the usual fashion.

If you specify both **inline** and **extern** in the function definition, the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function and hadn't defined it.

This combination of **inline** and **extern** has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking **inline** and **extern**) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

Assembler Instructions with C Expression Operands

In an assembler instruction using **asm**, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here's how to use the 68881's **fsinx** instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here **angle** is the C expression for the input operand while **result** is that of the output operand. Each has **f** as its operand constraint, saying that a floating point register is required. The = in **=f** indicates that the operand is an output; all output operands' constraints must use =. The constraints use the same language used in the machine description.

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to the maximum number of operands in any instruction pattern in the machine description.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It doesn't parse the assembler instruction template and doesn't know what it means, or whether it's valid assembler input. The extended **asm** feature is most often used for machine instructions that the compiler itself doesn't know exist.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended **asm** doesn't support input-output or read-write operands. For this reason the constraint character **+**, which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) **combine** instruction with **bar** as its read-only source operand and **foo** as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint **0** for operand 1 says that it must occupy the same location as operand 0. A digit in the constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that **foo** is the value of both operands isn't enough to guarantee that they will be in the same place in the generated assembler code. The following wouldn't work:

```
asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of **foo** in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to **foo**'s own address). Of course, since the register for operand 1 isn't even mentioned in the assembler code, the result won't work, but GNU CC can't tell that.

Unless an output operand has the **&** constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use **&** for each output operand that may not overlap an input.

You can put multiple assembler instructions together in a single **asm** template, separated either with newlines (written as **\n**) or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons, and all UNIX assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here's an example of multiple instructions in a template; it assumes that the subroutine **_foo** accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the **asm** construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
     : "g" (result)
     : "g" (input));
```

This assumes that your assembler supports local labels, as the GNU assembler and most UNIX assemblers do.

Usually the most convenient way to use these **asm** instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)        \
({ double __value, __arg = (x);    \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));   \
   __value; })
```

Here the variable **__arg** is used to make sure that the instruction operates on a proper **double** value, and to accept only those arguments **x** which can convert automatically to a **double**.

Another way to make sure the instruction operates on the correct data type is to use a cast in the **asm**. This is different from using a variable **__arg** in that it converts more different types. For example, if the desired type were **int**, casting the argument to **int** would accept a pointer with no complaint, while assigning the argument to an **int** variable named **__arg** would warn about using a pointer unless the caller explicitly casts it.

If an **asm** has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This doesn't mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an **asm** instruction from being deleted, moved significantly, or combined, by writing the keyword **volatile** after the **asm**. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands won't be deleted or moved significantly, regardless, unless it's unreachable.

Note that even a volatile **asm** instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile **asm** instructions to remain perfectly consecutive. If you want consecutive output, use a single **asm**.

It's a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

Additional Information about GNU CC

This section describes a few areas that commonly cause problems for users of GNU CC, and points out incompatibilities between GNU C and some other existing versions of C.

Known Causes of Trouble with GNU CC

Here are some of the things that have caused trouble for people installing or using GNU CC.

- Users often think it's a bug when GNU CC reports an error for code like this:

  ```
  int foo (short);

  int foo (x)
       short x;
  {. . .}
  ```

  The error message is correct: this code really is erroneous, because the old-style non-prototype definition passes subword integers in their promoted types. In other words, the argument is really an **int**, not a **short**. The correct prototype is this:

  ```
  int foo (int);
  ```

- Users often think it's a bug when GNU CC reports an error for code like this:

  ```
  int foo (struct mumble *);

  struct mumble { . . . };

  int foo (struct mumble *x)
  { . . . }
  ```

  This code really is erroneous, because the scope of **struct mumble** the prototype is limited to the argument list containing it. It doesn't refer to the **struct mumble** defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

  But in the definition of **foo**, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype don't match, and you get an error.

  This behavior may seem silly, but it's what the ANSI standard specifies. It's easy enough for you to make your code work by moving the definition of **struct mumble** above the prototype. It's not worth being incompatible with ANSI C just to avoid an error for the example shown above.

Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The **-traditional** option eliminates most of these incompatibilities—but not all—by telling GNU C to behave like the other C compilers.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call **mktemp()** with a string constant argument. The function **mktemp()** always alters the string its argument points to.

  Another consequence is that **sscanf()** doesn't work on some systems when passed a string constant as its format control string or input. This is because **sscanf()** incorrectly tries to write into the string constant. This is also true of **fscanf()** and **scanf()**.

  The best solution to these problems is to change the program to use char-array variables with initialization strings for these purposes instead of string constants. But if this isn't possible, you can use the **-fwritable-strings** flag, which directs GNU CC to handle string constants the same way most C compilers do. **-traditional** also has this effect, among others.

- GNU CC doesn't substitute macro arguments when they appear inside string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) "a"
  ```

  will produce output **"a"** regardless of what the argument **a** is.

  The **-traditional** option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use **setjmp()** and **longjmp()**, the only automatic variables guaranteed to remain valid are those declared **volatile**. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo ()
  {
    int a, b;

    a = fun1 ();
    if (setjmp (j))
      return a;

    a = fun2 ();
    /* @r{longjmp (j) may occur in fun3.} */
    return a + fun3 ();
  }
  ```

  Here **a** may or may not be restored to its first value when the **longjmp()** occurs. If **a** is allocated in a register, its first value is restored; otherwise, it keeps the last value stored in it.

  If you use the **-W** option with the **-O** option, you'll get a warning when GNU CC thinks such a problem might be possible.

  The **-traditional** option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call **setjmp()**. This results in the behavior found in traditional C compilers.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, an **extern** declaration affects all the rest of the file even if it happens within a block.

  The **-traditional** option directs GNU C to treat all **extern** declarations as global, like traditional compilers.

- In traditional C, you can combine **long**, etc., with a **typedef** name, as shown here:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ANSI C, this isn't allowed: **long** and other type modifiers require an explicit **int**. Because this criterion is expressed by Bison grammar rules rather than C code, the **-traditional** flag cannot alter it.

- PCC allows **typedef** names to be used as function parameters. The difficulty described immediately above applies here too.

- PCC allows whitespace in the middle of compound assignment operators such as **+=**. GNU CC, following the ANSI standard, doesn't allow this. The difficulty described immediately above applies here too.

- GNU CC will flag unterminated character constants inside preprocessor conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GNU CC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by **/* . . . */**. However, **-traditional** suppresses these error messages.

- When compiling functions that return **float**, PCC converts it to a **double**. GNU CC actually returns a **float**. If you are concerned with PCC compatibility, you should declare your functions to return **double**.

- When compiling functions that return structures or unions, GNU CC output code normally uses a method different from that used on most versions of UNIX. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special fixed register, but on some machines it's passed on the stack). The machine-description macros **STRUCT_VALUE** and **STRUCT_INCOMING_VALUE** tell GNU CC where to pass this address.

By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. GNU CC doesn't use this method because it's slower and nonreentrant.

On some newer machines, PCC uses a reentrant convention for all structure and union returning. GNU CC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

Legal Considerations

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

4. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

5. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

6. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not

normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

7. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

8. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

9. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

10. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

11. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

12. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

13. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

14. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

15. This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

16. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

17. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

18. Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

19. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

20. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES

OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

21. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.*
> Copyright (C) 19*yy* *name of author*
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19*yy name of author*
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands **show w** and **show c** should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than **show w** and **show c**; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here's a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

12 *The GNU C Preprocessor*

12 The GNU C Preprocessor

The GNU C preprocessor is a macro processor the C compiler uses to transform your program before actual compilation. It's called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

The C preprocessor provides the following four facilities:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define and use macros, which are abbreviations for arbitrary fragments of C code. The C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessor commands, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in their implementation details. This section describes the GNU C preprocessor, which provides a superset of the features of ANSI-standard C.

ANSI-standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. To get ANSI-standard C you would use the options **-trigraphs**, **-undef**, and **-pedantic**, although in practice the consequences of having strict ANSI Standard C may make it undesirable to do this. See the section "Invoking the C Preprocessor" for more information.

Global Transformations

Most C preprocessor features are inactive unless you give specific commands to request their use. But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands:

- C comments (and Objective C comments) are replaced with single spaces.
- Backslash-newline sequences are deleted. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see the section "Predefined Macros").

The first two transformations are done *before* nearly all other parsing and before preprocessor commands are recognized. Thus, for example, you can split a line cosmetically with backslash-newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent to **#define FOO 1020**. You can even split an escape sequence with backslash-newline. For example, you can split **"foo\bar"** between the backslash and the **b** to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a backslash can be inserted in a string constant as an ordinary character by writing a double backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C doesn't allow newlines in string constants, so this isn't considered a problem.)

There are a few exceptions to all three transformations:

- C comments and predefined macro names aren't recognized inside an **#include** command in which the file name is delimited with < and >.

- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule rather than an exception.)
- Backslash-newline may not safely be used within an ANSI trigraph (trigraphs are converted before backslash-newline is deleted). If you write what looks like a trigraph with a backslash-newline inside, the backslash-newline is deleted as usual, but it is then too late to recognize the trigraph.

  This exception is relevant only if you use the **-trigraphs** option to enable trigraph processing.

Preprocessor Commands

Most preprocessor features are active only if you use preprocessor commands to request their use.

Preprocessor commands are lines in your program that start with #. The # is followed by an identifier that's the command name. For example, **#define** is the command that defines a macro. White-space characters are allowed before and after the #.

The set of valid command names is fixed. Programs can't define new preprocessor commands.

Some command names require arguments; these make up the rest of the command line and must be separated from the command name by one or more white-space characters. For example, **#define** must be followed by a macro name and the intended expansion of the macro.

A preprocessor command normally can't be more than one line. It may be split cosmetically with backslash-newline, but that has no effect on its meaning. Comments containing newlines can also divide the command into multiple lines, but the comments are changed to spaces before the command is interpreted. The only way a significant newline can occur in a preprocessor command is within a string constant or character constant. (Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing newlines.)

The # and the command name can't come from a macro expansion. For example, if **foo** is defined as a macro expanding to **define**, that doesn't make **#foo** a valid preprocessor command.

Header Files

Header files can contain C declarations and macro definitions that are to be shared by more than one source file. You request the inclusion of a header file in a source file by using the C preprocessor command **#include** (or more typically in the NeXTSTEP environment, the Objective C preprocessor command **#import**).

Uses of Header Files

Header files serve two kinds of purposes:

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions, all or most of which are needed in several different source files, it's a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when recompiled.

By convention, names of header files end with the extension ".h".

The #include Command

Both user and system header files are included using the preprocessor command **#include**. It has three variants:

#include <*file*>

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option **-I** (see the section "Invoking the C Preprocessor"). The option **-nostdinc** inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of **#include** is slightly special because comments are not recognized within the <*file*> argument. Thus, in **#include <x/*y>** the **/*** doesn't start a comment and the command specifies inclusion of a system header file named **x/*y**. (Of course, a header file with such a name is unlikely to exist on a UNIX-based system, where shell wildcard features would make it hard to manipulate.)

The *file* argument may not contain a > character, although it may contain a < character.

#include "*file*"

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is tried first because it's presumed to be the location of the files of the program being compiled. (If the **-I-** option is used, the special treatment of the current directory is inhibited.)

The *file* argument may not contain " characters. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, **#include "x\n\\y"** specifies a file name containing three backslashes. It isn't clear why this behavior is ever useful, but the ANSI standard specifies it.

#include *anything else*

This variant is called a computed **#include**. Any **#include** command whose argument doesn't fit the above two forms is a computed **#include**. The text *anything else* is checked for macro calls, which are expanded. When this is done, the result must fit one of the above two variants.

This feature allows you to define a macro that controls the file name to be used at a later point in the program. One application of this is to allow a site-configuration file for your program to specify the names of the system header files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

The **#include** command directs the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor will contain the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** command. Included files can themselves contain **#include** commands to include other files.

Included files are not limited to declarations and macro definitions, although those are the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

The line following the **#include** command is always treated as a separate line by the C preprocessor, even if the included file lacks a final newline.

Note: The Objective C language equivalent of **#include** is **#import**; the only difference is that **#import** doesn't include a file more than once, no matter how many **#import** commands try to include it. You should feel free to use **#import** in your code, but be aware that it isn't defined as part of ANSI-standard C.

Multiple Inclusion of Header Files

Very often one header file includes another, which can result in a certain header file being included more than once. This may lead to errors if the header file defines structure types or typedefs, and in any event is wasteful. For these reasons, you should try to avoid multiple inclusion of a header file.

The standard way to prevent multiple inclusion of a file is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef __FILE_FOO_SEEN__
#define __FILE_FOO_SEEN__
the entire file
#endif /* __FILE_FOO_SEEN__ */
```

The macro **__FILE_FOO_SEEN__** indicates that the file has been included once already; its name begins with __ to avoid conflicts with user programs, and it contains the name of the file and some additional text to avoid conflicts with other header files.

Alternatively (if compatibility with non-NeXT platforms isn't an issue), you can ensure that each file is included only once simply by using the Objective C **#import** command instead of the **#include** command.

Precompiled Header Files

A precompiled header is a C header file that has been preprocessed and parsed, thereby improving compile time and reducing symbol table size. The macros and external declarations from the original header are sorted to enable fast lookup. A new implementation of the C preprocessor can use precompiled headers in place of standard headers.

In most cases, the use of precompiled headers is transparent. Precompiled headers are simple enough to use that most projects require no conversion at all, or can be converted in a day or less.

Using Precompiled System Header Files

The precompiled version of a header file has a ".p" extension, rather than the standard ".h" extension. You should *not* refer to **appkit.p** in your source files; just use **appkit.h** and the preprocessor will use the precompiled form if it's available and appropriate.

When the preprocessor encounters an include directive, it automatically looks for a precompiled version of the header. If one is found, it checks whether the context is equivalent to the context in which the precompiled header was built—if it is, the precompiled header is used. However, if any of the following problems occur, the non-precompiled form is included instead:

- A header which was included by the precompiled header could not be found in the filesystem to verify its modification time, or the modification time did not match. In practice, this never occurs for precompiled headers that are part of the release, and occurs only rarely when programmers build their own precompiled headers.
- A macro was defined when the precompiled header was built, but is not defined in the current context. This is only a problem if the macro was actually referenced somewhere in the precompiled header.
- A macro was undefined when the precompiled header was built, but is defined in the current context. This is only a problem if there might have been an invocation of the macro in the precompiled header.

Compile-time warnings (described at the end of this file) indicate the nature of any problems that occur. However, you need to compile with the **-Wprecomp** option in order for these warnings to be displayed.

If you're developing a small project, you don't need to bother building you own precompiled headers—just use the precompiled system headers **appkit.p** and **mach.p**. It's easy to create your own precompiled headers if you wish to do so, however, as described in the next section.

Creating Your Own Precompiled Header Files

You create a precompiled header by passing the new **-precomp** switch to **cc**. Depending on the context(s) in which the header is used, **-D** switches should also be passed to **cc**, as explained below.

```
% cc -precomp foo.h -o foo.p
```

We say a header is "context dependent" if the definitions in the header may change depending on the context in which it is included. Most uses of conditional compilation and macro expansions cause context dependence. For instance, the following header is context dependent:

```
#ifdef DEBUG
int a;
#else
int b;
#endif
```

The context at any point is determined by the macros that are defined there. A precompiled header must be created in a context equivalent to that where it is used. By passing switches to the preprocessor, any set of macros can be predefined, creating a context in which the precompiled header is built. This is done by passing a **-D** switch for each macro in the context.

A precompiled header built from "system headers" typically requires no **-D** switches, because programmers usually include system headers in a context independent way. For instance, the public appkit headers contain almost no preprocessor conditionals; clients cannot change declarations in headers by defining macros. So the command to build a precompiled header from **appkit.h** is:

```
% cc -precomp appkit.h -o appkit.p
```

But if you must use a header **bar.h** in a context where FOO is defined, you should build the precompiled header as follows:

```
% cc -precomp -DFOO bar.h -o bar.p
```

You should also pass any preprocessor switches, such as **-I**, that you use in your project.

By making precompiled headers bigger (that is, containing more headers), a given C file may include fewer precompiled headers, and will generally compile faster. However, the bigger a precompiled header is, the more likely that name conflicts will occur.

For example, if you were to combine all the headers for a project, including system headers, into a single precompiled header, it is quite possible that there would be a name conflict. There may be a macro defined that happens to match one of your local identifiers, or there may be a public struct declared that happens to match one of your private struct names. Such conflicts manifest themselves as preprocessing errors, syntax errors, or semantic errors. The conflicts may be resolved by renaming identifiers, or removing a conflicting header from the precompiled header.

Another disadvantage to big precompiled headers is file dependencies. If all of the C files in a project depend on a single precompiled header which in turn depends on all headers in the project, then changing a header requires recompilation of the entire project. A better approach is to build a precompiled header containing all the system headers used by a project, and perhaps also a separate precompiled header for the local headers in the project. We recommend that during development, while local headers are changing, precompiled headers be used only for system files. When local headers have stabilized, they may be combined into a precompiled header.

A precompiled header is dependent on all the files it includes. A Make dependency rule can be constructed similar to the way rules are constructed for source files (see the Make rule for **depend:** in **Makefile.common**). The following rule builds a precompiled header from a header:

```
.h.p:
    cc -precomp $(CFLAGS) $*.h $*.p
```

A precompiled header records absolute path names for all the headers that went into it. These paths are then checked when the precompiled header is used. Therefore a precompiled header should be built in the same directory in which it is to be used, and all the headers that went into the precompiled header must not be moved or modified.

Troubleshooting

The Release 3.0 preprocessor and parser are required in order to use precompiled headers, and there are several incompatibilities with the Release 2.0 preprocessor and parser. For example, preprocessing errors and syntax errors are in a slightly different format.

Only rarely will you have trouble building a precompiled header. The most common problem you might encounter is that the header doesn't parse; this is often because the header does not include other headers it depends on, so that there are undefined types. Another typical problem is conflicting definitions, which can be solved by renaming identifiers or removing a header from the precompiled header.

The following list describes the compile-time warnings that may occur when using a precompiled header (currently you must compile with the **-Wprecomp** option in order for these warnings to be displayed):

- **could not use precompiled header '*header*.p'**

  The precompiled header could not be used for one of the reasons below.

- **macro '*macro*' undefined**

  The macro was defined when the precompiled header was built, but is not defined in the current context.

- **macro '*macro*' defined**

  The macro was undefined when the precompiled header was built, but is defined in the current context. This error can often be avoided by importing precompiled headers in the source file before any other headers.

- **macro '*macro*' defined by '*header*.p' conflicts with precomp**

  A previously included precompiled header defines a macro differently than does the current precompiled header being processed.

- **macro '*macro*' defined on command line conflicts with precomp**

  Similar to the previous warning, except that the earlier definition of the macro occurred on the command line.

- **macro 'macro' redefined, locations of the conflict are:**
  ***header1*.h:23**
  ***header2*.h:47 (within the precompiled header)**

  The macro has been defined in two different ways in two different precompiled headers

- **#ifdef '*SYM*' not defined when precompiled**

  A symbol was defined for the inclusion of this precompiled header, but was not when the header was precompiled. Since this symbol is used in an #ifdef, the precompiled header does not contain all the source code desired by the including context.

- **'*header*.h' has different date than in precomp**

  The modification time of the header on the disk does not match the modification time of the header when the precompiled header was built.

- **could not find '*header*.h'**

  The header which was included by the precompiled header could not be found on the disk to verify its modification time.

- **could not use precomp '*header*.p' (incorrect version)**

  It was discovered that the version of the referenced precompiled header is incompatible with the compiler, possibly signifying a corrupt or obsolete **header.p.**

- **explicit reference to precompiled 'header.p' failed**

  Although the inclusion of headers with a ".p" suffix is discouraged due to portability considerations, it is legal to explicitly reference precompiled headers. The above error is generated if the precompiled header could not be typechecked properly.

Macros

A macro is an abbreviation you define once and then use later. This section describes some important features associated with macros in the C preprocessor.

Simple Macros

A simple macro is a kind of abbreviation—it's a name that stands for a fragment of code. Simple macros are sometimes referred to as *manifest constants.*

Before you can use a macro, you must define it explicitly with the **#define** command. **#define** is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named **BUFFER_SIZE** as an abbreviation for the text **1020.** With this definition in effect, the C preprocessor would expand the following statement

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

to

```
foo = (char *) xmalloc (1020);
```

The definition must be a single line; however, it may not end in the middle of a multiline string constant or character constant.

For readability, uppercase is used for macro names by convention. Programs are easier to read when it's possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line (although you can always split a long macro definition cosmetically with backslash-newline). There's one exception: Newlines can be included in the macro definition if they're within a string or character constant. It isn't possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain newlines (which make no difference, since the comments are entirely replaced with spaces regardless of their contents).

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance, and the body need not resemble valid C code. (Of course, you might get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces as output:

```
foo = X;
bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macros continues. Therefore, the macro body can contain other macros. For example, after the following definitions

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name **TABLESIZE** when used in the program would go through two stages of expansion, resulting ultimately in **1020**.

This isn't the same as defining **TABLESIZE** to be **1020**. The **#define** for **TABLESIZE** uses exactly the body you specify—in this case, **BUFSIZE**—and doesn't check to see whether it too is the name of a macro. It's only when you use **TABLESIZE** that the result of its expansion is checked for more macro names. See the section "Cascaded Use of Macros."

Macros that Take Arguments

A simple macro always stands for exactly the same text, each time it's used. Macros can be more flexible when they accept arguments. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition.

To define a macro that takes arguments, you use the **#define** command with a list of parameters in parentheses after the name of the macro. The parameters may be any valid C identifiers separated by commas (and optionally, by white-space characters). The left parenthesis must follow the macro name immediately, with no space in between.

For example, here's a macro that computes the minimum of two numeric values:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

Note that this isn't the best way to define a "minimum" macro in GNU C (see the section "Duplication of Side Effects" for more information).

To use a macro that takes arguments, you write the name of the macro followed by a list of arguments in parentheses, separated by commas. The number of arguments you give must match the number of parameters in the macro definition. The following examples show the use of the macro **min**:

```
min (1, 2)
min (x + 28, *p)
```

The expansion text of the macro depends on the arguments you use. Each of the macro's parameters is replaced, throughout the macro definition, with the corresponding argument. Using the same macro **min** defined above, **min (1, 2)** expands to

```
((1) < (2) ? (1) : (2))
```

where **1** has been substituted for **X** and **2** for **Y**.

Likewise, **min (x + 28, *p)** expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the arguments must balance; a comma within parentheses doesn't end an argument. However, there's no requirement for brackets or braces to balance; thus, if you want to supply

```
array[x = y, x + 1]
```

as an argument, you would write it as

```
array[(x = y, x + 1)]
```

After the arguments are substituted into the macro body, the entire result is appended to the front of the remaining input, and the check for macros continues. Therefore, the arguments can contain other macros, either with or without arguments, or even the same macro. The macro body can also contain other macros. For example, **min (min (a, b), c)** expands into

```
((((a) < (b) ? (a) : (b))) < (c)
    ? (((a) < (b) ? (a) : (b)))
    : (c))
```

Line breaks shown here for clarity wouldn't actually be generated.

If you use the macro name followed by something other than a left parenthesis (after ignoring any spaces, tabs, and comments that follow), it isn't considered a macro invocation, and the preprocessor doesn't change what you've written. Therefore, it's possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an argument list follows) or the variable or function (if an argument list doesn't follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro. For example, you can use a function named **min** in the same source file that defines the macro. If you write **&min** with no argument list, you refer to the function. If you write **min (x, bb)**, with an argument list, the macro is expanded. If you write **(min) (a, bb)**, where the name **min** isn't followed by a left parenthesis, the macro isn't expanded; rather, the function **min** is called.

A name can't be defined as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and the rest of the name is taken to be the expansion. The reason for this is that it's often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this (which defines **FOO** to take an argument and expand into minus the reciprocal of that argument)

```
#define FOO(x) - 1 / (x)
```

or this (which defines **FOO** to take no argument and always expand into **(x) - 1 / (x)**):

```
#define FOO (x) - 1 / (x)
```

It matters only in the macro definition whether there's a space before the left parenthesis; when you use the macro, it doesn't matter if there are spaces there or not.

Predefined Macros

Several standard macros are predefined, some by ANSI C and some as extensions. Their names all start and end with double underscores.

The following predefined macros are part of the ANSI C standard:

__FILE__

This macro expands to the name of the current input file, in the form of a C string constant.

__BASE_FILE__

This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

__LINE__

This macro expands to the current input line number, in the form of a decimal integer constant. (Note that although this is considered a predefined macro, its definition changes with each new line of source code.)

This and **__FILE__** are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example:

```
fprintf (stderr,
        "Internal error: negative string length "
        "%d at %s, line %d."
        length, __FILE__, __LINE__);
```

An **#include** command changes the expansions of **__FILE__** and **__LINE__** to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the **#include** command, the expansions of **__FILE__** and **__LINE__** revert to the values they had before the **#include** (but **__LINE__** is then incremented by one as processing moves to the line after the **#include**).

The expansions of both **__FILE__** and **__LINE__** are altered if a **#line** command is used. See the section "Combining Source Files."

__DATE__

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains 15 characters and looks like "Tue Jun 02 1992".

__TIME__

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains 12 characters and looks like "23:59:01 EDT".

__STDC__

This macro expands to the constant 1, to signify that this is ANSI-standard C. (Whether that's actually true depends on what C compiler will operate on the output from the preprocessor.)

The following predefined macros are GNU C extensions to the ANSI C standard:

__GNUC__

This macro is defined if and only if this is GNU C. Moreover, it's defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, **__GNUC__** is undefined.

__STRICT_ANSI__

This macro is defined if and only if the **-ansi** switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define traditional UNIX constructs that are incompatible with ANSI C.

__VERSION__

This macro expands to a string describing the version number of the compiler. The string is normally a sequence of decimal numbers separated by periods, such as **"1.18"**. The main use of this macro is to incorporate the version number into a string constant.

__OPTIMIZE__

This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It's unwise to refer to or test the definition of this macro unless you make sure that programs will execute with the same effect regardless.

__CHAR_UNSIGNED__

This macro is defined if and only if the data type **char** is unsigned on the target machine. Its purpose is to cause the standard header file **limit.h** to work correctly. It's bad practice to refer to this macro yourself; instead, refer to the standard macros defined in **limit.h**.

The following macros are defined in NeXTSTEP:

__OBJC__

This macro is defined when compiling Objective C ".m" files.

__GNU__

This macro is defined when compiling ".m", ".c", or ".s" files.

__ASSEMBLER__

This macro is defined when compiling ".s" files.

__STRICT_BSD__

This macro is defined if and only if the **-bsd** switch was specified when GNU C was invoked.

__MACH__

This macro is defined if Mach system calls are supported.

Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This section lists some that are useful on NeXTSTEP computers.

Some nonstandard predefined macros describe the operating system in use. For example:

unix	Predefined on UNIX systems.
BSD	Predefined on versions of Berkeley UNIX 4.3BSD.

Other nonstandard predefined macros describe the kind of CPU. For example:

mc68000 Predefined on most computers whose CPU is a Motorola 68000, 68010, 68020, 68030, or 68040.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example:

NeXT Predefined on a NeXT computer.

These predefined symbols aren't only nonstandard, they're contrary to the ANSI standard because their names don't start with underscores. The **-ansi** option, which requests complete support for ANSI C, inhibits the definition of these predefined symbols.

This tends to make **-ansi** useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with **-ansi**. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it will run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding __ at the beginning and end. Thus, the symbol **__vax__** would be available on a VAX, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled (when **cpp** is itself compiled) by the macro **CPP_PREDEFINES**, which should be a string containing -D options, separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

This macro is usually specified in **tm.h**.

Stringification

"Stringification" means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying **foo (z)** results in **"foo (z)"**.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character # before the name specifies stringification of the corresponding argument when it's substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no #.

Here's an example of a macro definiiion that uses stringification:

```
#define WARN_IF(EXP)   \
do { if (EXP) fprintf (stderr, "Warning: " #EXP "\n"); }
    while (0)
```

Here the argument for **EXP** is substituted once as given, into the **if** statement, and once as stringified, into the argument to **fprintf**. The **do** and **while (0)** make it possible to write **WARN_IF (ARG);** (see the section "Swallowing the Semicolon").

The stringification feature is limited to transforming one macro argument into one string constant: There's no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI-standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies **EXP**'s argument into a separate string constant, resulting in text like

```
do { if (x == 0) fprintf (stderr, "Warning: " "x == 0" "\n"); }
    while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing:

```
do { if (x == 0) fprintf (stderr, "Warning: x == 0\n"); }
    while (0)
```

Stringification in C involves more than putting double quotes around the fragment; it's necessary to put backslashes in front of all double quotes, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying **p = "foo\n";** results in **"p = \"foo\\n\";"**. However, backslashes that aren't inside string or character constants aren't duplicated: **\n** by itself stringifies to **"\n"**.

White-space characters (including comments) in the text being stringified are handled according to the following rules:

- All leading and trailing white-space characters are ignored.
- Any sequence of white-space characters in the middle of the text is converted to a single space in the stringified result.

Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an argument to the macro can be concatenated with another argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator ## in the macro body. When the macro is invoked, arguments are substituted. Then all ## operators are deleted, along with any white-space characters next to them (including white-space characters that are part of an argument). The result is to concatenate the syntactic tokens on either side of the ##.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
    char *name;
    void (*function) ();
};

struct command commands[] =
{
    { "quit", quit_command},
    { "help", help_command},
    . . .
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro that takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with "_command":

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
    COMMAND (quit),
    COMMAND (help),
    . . .
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It's also possible to concatenate two numbers (or a number and a name, such as **1.5** and **e3**) into a number. Also, multicharacter operators such as **+=** can be formed by concatenation. In some cases it's even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with **x** on one side and **+** on the other isn't meaningful because those two characters can't fit together in any lexical unit of C. Although the ANSI standard says that such an attempt at concatenation is undefined, the GNU C preprocessor handles it as follows: it puts the **x** and **+** side by side with no particular special results.

The C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating **/** and *****: the **/*** sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. You can freely use comments next to a **##** in a macro definition, or in arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

Undefining Macros

To undefine a macro means to cancel its definition. This is done with the **#undef** command. **#undef** is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it's treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;
x = FOO;
```

In this example, **FOO** must be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of **#undef** command will cancel definitions with arguments or definitions that don't expect arguments. The **#undef** command has no effect when used on a name not currently defined as a macro.

Redefining Macros

Redefining a macro means defining (with **#define**) a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see the section "Header Files"), so they're accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it's useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with **#undef** before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace previously. Remember, comments count as whitespace.

Pitfalls and Subtleties of Macros

This section describes some special rules that apply to macros and macro expansion, and points out certain cases in which the rules have counterintuitive consequences that you must watch out for.

Improperly Nested Constructs

Recall that when a macro is invoked with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macros.

It's possible to piece together a macro invocation coming partially from the macro body and partially from the arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand **call_with_1 (double)** into **(2*(1))**.

Macro definitions don't have to have balanced parentheses. By writing an unbalanced left parenthesis in a macro body, it's possible to create a macro invocation that begins inside the macro body but ends outside it. For example:

```
#define strange(file) fprintf (file, "%s %d",
. . .
strange(stderr) p, 35)
```

This bizarre example expands to

```
fprintf (stderr, "%s %d", p, 35)
```

Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name has parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. This section discusses why it's best to write macros that way.

Suppose you define a macro

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many **int**'s are needed to hold a certain number of **chars**.) Then suppose it's used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which doesn't do what's intended. The operator-precedence rules of C make this equivalent to:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as follows provides the desired result:

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

However, unintended grouping can happen in another way. Consider **sizeof ceil_div(1, 2)**. This has the appearance of a C expression that would compute the size of the type of **ceil_div (1, 2)**, but in fact it means something very different. Here's what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by 2. The precedence rules have put the division outside the **sizeof()** when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here's the recommended way to define **ceil_div**:

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

Swallowing the Semicolon

Often it's desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, which advances a pointer across space characters:

```
#define SKIP_SPACES (p, limit)  \
{ register char *lim = (limit); \
    while (p != lim) {  \
        if (*p++ != ' ') {  \
                p-; break; }}}
```

Here backslash-newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would appear if not part of a macro definition.

An invocation of this macro might be **SKIP_SPACES (p, lim)**. Strictly speaking, the invocation expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward:

```
SKIP_SPACES (p, lim);
```

But this can cause trouble before **else** statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
    SKIP_SPACES (p, lim);
else . . .
```

The presence of two statements—the compound statement and a null statement—in between the **if** condition and the **else** makes invalid C code.

The definition of the macro **SKIP_SPACES** can be altered to solve this problem, using a **do ... while** statement:

```
#define SKIP_SPACES (p, limit)  \
do { register char *lim = (limit);  \
    while (p != lim) {  \
        if (*p++ != ' ') {  \
                p-; break; }}}  \
while (0)
```

Now **SKIP_SPACES (p, lim);** expands into one statement:

```
do {. . .} while (0);
```

Duplication of Side Effects

Many C programs define a macro **min** (for "minimum"), like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect (as shown here)

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where **x + y** has been substituted for **X** and **foo (z)** for **Y**.

The function **foo** is used only once in the statement as it appears in the program, but the expression **foo (z)** has been substituted twice into the macro expansion. As a result, **foo** might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. Therefore **min** is an "unsafe" macro.

One way to solve this problem is to define **min** in a way that computes the value of **foo (z)** only once. The C language offers no standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                          \
(@{ typeof (X) __x = (X), __y = (Y);    \
   (__x < __y) ? __x : __y; @})
```

If you don't wish to use GNU C extensions, the only solution is to be careful when using the macro **min**. For example, you can calculate the value of **foo (z)**, save it in a variable, and use that variable in **min**:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
. . .
{
     int tem = foo (z);
     next = min (x + y, tem);
}
```

Self-Referential Macros

A self-referential macro is one whose name appears in its definition. A special feature of ANSI-standard C is that the self-reference isn't considered a macro invocation. It's passed into the preprocessor output unchanged.

Consider the following example (assume that **foo** is also a variable in your program):

```
#define foo (4 + foo)
```

Following the ordinary rules, each reference to **foo** will expand into **(4 + foo)**; then this will be rescanned and will expand into **(4 + (4 + foo))**; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at **(4 + foo)**. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of **foo** wherever **foo** is referred to.

In most cases, it's a bad idea to take advantage of this feature. A person reading the program who sees that **foo** is a variable won't expect that it's a macro as well. The reader will come across the identifier **foo** in the program and think its value should be that of the variable **foo**, whereas in fact the value is 4 greater.

The special rule for self-reference applies also to indirect self-reference. This is the case where a macro X expands to use a macro **y**, and **y**'s expansion refers to the macro **x**. The resulting reference to **x** comes indirectly from the expansion of **x**, so it's a self-reference and isn't further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

x would expand into **(4 + (2 * x))**.

But suppose **y** is used elsewhere, not from the definition of **x**. Then the use of **x** in the expansion of **y** isn't a self-reference because **x** isn't in progress. So it does expand. However, the expansion of **x** contains a reference to **y**, and that's an indirect self-reference now because **y** is in progress. The result is that **y** expands to **(2 * (4 + y))**.

Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted arguments, is scanned over again for macros to be expanded.

What really happens is more subtle: First each argument text is scanned separately for macros. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the arguments are scanned twice to expand macros in them.

Most of the time, this has no effect. If the argument contained any macros, they're expanded during the first scan. The result therefore contains no macros, so the second scan doesn't change it. If the argument were substituted as given, with no prescan, the single remaining scan would find the same macros and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an argument of another macro (see the section "Self-Referential Macros" above); the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this isn't what happens. The self-references that don't expand in the first scan are marked so that they won't expand in the second scan either.

The prescan isn't done when an argument is stringified or concatenated. (More precisely, stringification and concatenation use the argument as written, in unprescanned form. The same argument would be used in prescanned form if it's substituted elsewhere without stringification or concatenation.) Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to **"foo"**. Once more, prescan has been prevented from having any noticeable effect.

The prescan does make a difference in three special cases:

- Nested invocations of a macro
- Macros that invoke other macros that stringify or concatenate
- Macros whose expansions contain unshielded commas

Nested invocations of a macro occur when a macro's argument contains an invocation of that very macro. For example, if **f** is a macro that expects one argument, **f (f (1))** is a nested pair of invocations of **f**. The desired expansion is made by expanding **f (1)** and substituting that into the definition of **f**. The prescan causes the expected result to happen. Without the prescan, **f (1)** itself would be substituted as an argument, and the inner use of **f** would appear during the main scan as an indirect self-reference and wouldn't be expanded. Here, the prescan cancels an undesirable side effect of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. For example:

```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))

bar(foo)
```

We would like **bar(foo)** to turn into **(1 + (foo))**, which would then turn into **(1 + (a,b))**. But instead, **bar(foo)** expands into **lose(a,b)**, and you get an error because **lose** requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
#define bar(x) lose((x))
```

The problem is more serious when the operands of the macro aren't expressions (for example, when they are statements). Then parentheses are unacceptable because they would make for invalid C code:

```
#define foo { int a, b; ... }
```

In GNU C you can shield the commas using the **({ ... })** construct, which turns a compound statement into an expression:

```
#define foo ({ int a, b; ... })
```

Or you can rewrite the macro definition to avoid such commas:

```
#define foo { int a; int b; ... }
```

There's also one case where prescan is useful. It's possible to use prescan to expand an argument and then stringify it—if you use two levels of macros. Let's add a new macro **xstr** to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands to **"4"**, not **"foo"**. The reason for the difference is that the argument of **xstr** is expanded at prescan (because **xstr** doesn't specify stringification or concatenation of the argument). The result of prescan then forms the argument for **str**. **str** uses its argument without prescan because it performs stringification; but it can't prevent or undo the prescanning already done by **xstr**.

Cascaded Use of Macros

A cascade of macros occurs when one macro's body contains a reference to another macro (a very common practice). For example:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This isn't at all the same as defining **TABLESIZE** to be **1020**. The **#define** for **TABLESIZE** uses exactly the body you specify—in this case, **BUFSIZE**—and doesn't check to see whether it too is the name of a macro.

It's only when you *use* **TABLESIZE** that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of **BUFSIZE** at some point in the source file. **TABLESIZE**, defined as shown, will always expand using the definition of **BUFSIZE** that's currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now **TABLESIZE** expands in two stages to **37**.

Inability to Define a Macro that Produces a # Character

You can't use the GNU C preprocessor to define macros that produce # characters. For instance, the following is illegal:

```
#define linkmacro(numBytes) link #numBytes,a6
```

Note that you can use the # character inside a string or character constant, as shown here:

```
#define PrintSharp() printf("#")
```

Macro Arguments inside String Constants

The GNU C preprocessor doesn't substitute macro arguments that appear inside string constants. For example, the following macro will produce the output **"a"** no matter what the argument **a** is:

```
#define foo(a) "a"
```

The **-traditional** option directs GNU CC to handle such cases (among others) in the traditional non-ANSI way.

Conditionals

In a macro processor, a conditional is a command that allows part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles an **if** statement in C, but it's important to understand the difference between them. The condition in an **if** statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it's operating on. The condition in a preprocessor conditional command is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

There are three reasons to use a conditional:

- A program may need to use different code depending on the target machine or operating system. In some cases, the code for one operating system may be erroneous on another operating system; for example, it might refer to library routines that don't exist on the other system. When this happens, it isn't enough to avoid executing the invalid code: Merely having it in the program makes it impossible to link the program and run it. With a preprocessor conditional, the offending code can be effectively excised from the program when it isn't valid.
- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data while the other doesn't.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it for future reference.

Most programs intended to run only on a NeXT computer won't need to use preprocessor conditionals.

Syntax of Conditionals

A conditional in the C preprocessor begins with a conditional command: **#if**, **#ifdef**, or **#ifndef**. These and a few related commands are described in the following sections.

The #if Command

The **#if** command in its simplest form consists of

> **#if** *expression*
> *conditional-text*
> **#endif** /* *expression* */

The comment following the **#endif** isn't required, but it makes the code easier to read. Such comments should always be used, except in short conditionals that aren't nested. (Although you can put anything at all after the **#endif** and it will be ignored by the GNU C preprocessor, only comments are acceptable in ANSI Standard C.)

expression is a C expression of type **int**, subject to stringent restrictions. It may contain:

- Integer constants, which are all regarded as **long** or **unsigned long**.
- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type **char** for these character constants; therefore, whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats **char** as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.
- Character constants. The GNU C preprocessor uses the C data type **char** for these character constants.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, **&&**, and ||.
- Identifiers that aren't macros, which are all treated as 0.
- Macro invocation. All macros in the expression are expanded before actual computation of the expression's value begins.

sizeof operators and **enum**-type values aren't allowed. **enum**-type values, like all other identifiers that aren't taken as macro invocations and expanded, are treated as 0.

The controlled text inside a conditional can include preprocessor commands. Then the commands inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the **#if**s and **#endif**s must balance.

The #else Command

The **#else** command can be added to a conditional to provide alternative text to be used if the condition is false:

> **#if** *expression*
> *text-if-true*
> **#else** /* not *expression* */
> *text-if-false*
> **#endif** /* not *expression* */

If *expression* is nonzero, *text-if-true* is included; then **#else** acts like a failing conditional and *text-if-false* is ignored. If *expression* is 0, the **#if** conditional fails and *text-if-false* is included.

The #elif Command

A common use of nested conditionals is to check for more than two possible alternatives:

```
#if X == 1
. . .
#else /* X != 1 */
#if X == 2
. . .
#else /* X != 2 */
. . .
#endif /* X != 2 */
#endif /* X != 1 */
```

The conditional command **#elif** (which stands for "else if") can be used to abbreviate this as follows:

```
#if X == 1
. . .
#elif X == 2
. . .
#else /* X != 2 and X != 1*/
. . .
#endif /* X != 2 and X != 1*/
```

Like **#else**, **#elif** goes in the middle of a **#if-#endif** pair and subdivides it; it doesn't require a matching **#endif** of its own. Like **#if**, the **#elif** command includes an expression to be tested.

The text following the **#elif** is processed only if the original **#if**-condition failed and the **#elif** condition succeeds. More than one **#elif** can go in the same **#if-#endif** group. Then the text after each **#elif** is processed only if the **#elif** condition succeeds after the original **#if** and any previous **#elif**'s within it have failed. **#else** is allowed after any number of **#elif**s, but **#elif** may not follow a **#else**.

Keeping Deleted Code for Future Reference

If you replace or delete part of the program but want to keep the old code around as a comment for future reference, you can simply put **#if 0** before it and **#endif** after it.

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced **#if** and **#endif**).

Conditionals and Macros

Conditionals are rarely useful except in connection with macros. A **#if** command whose expression uses no macros is equivalent to **#if 1** or **#if 0**; you may want to determine which one by computing the value of the expression yourself, thus simplifying the code. But when the expression uses macros, its value can vary from compilation to compilation.

For example, here's a conditional that tests the expression **BUFSIZE == 1020**, where **BUFSIZE** must be a macro:

```
#if BUFSIZE == 1020
    printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

The special operator **defined** may be used in **#if** expressions to test whether a certain name is defined as a macro. Either **defined NAME** or **defined (NAME)** is an expression whose value is 1 if NAME is defined as macro at the current point in the program, and 0 otherwise. For the **defined** operator it makes no difference what the definition of the macro is; all that matters is whether there's a definition. Thus, for example,

```
#if defined (vax)  defined (ns16000)
```

will include the following code if either **vax** or **ns16000** is defined as a macro.

If a macro is defined and later undefined with **#undef**, subsequent use of the **defined** operator will return 0, because the name is no longer defined. If the macro is defined again with another **#define**, **defined** will again return 1.

Conditionals that test just the definedness of one name are very common, so there are two special short conditional commands for this case:

- **#ifdef** *name* is equivalent to **#if defined** (*name*).
- **#ifndef** *name* is equivalent to **#if ! defined** (*name*).

Macro definitions can vary between compilations for any of the following reasons:

- Some macros are predefined on each kind of machine. For example, on a NeXT computer the name **NeXT** is a nonstandard predefined macro. On other machines, it isn't defined.

- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It's useful to test these macros with conditionals to avoid using a system feature on a machine where it isn't implemented.

- Macros are a common way for you to customize a program for different machines or applications. For example, the macro **BUFSIZE** might be defined in a configuration file for your program that's included as a header file in each source file. You would use **BUFSIZE** in a preprocessor conditional in order to generate different code depending on the chosen configuration.

- Macros can be defined or undefined with **-D** and **-U** command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See the section "Invoking the C Preprocessor."

The #error and #warning Commands

The **#error** command causes the preprocessor to report a fatal error. The rest of the line that follows **#error** is used as the error message.

You would use **#error** inside a conditional that detects a combination of parameters that you know the program doesn't support.

For example, if you know that the program won't run properly on a VAX, you might write

```
#ifdef vax
#error Won't work on Vaxen.  See comments at get_last_object.
#endif
```

Similarly, if you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with **#error**. For example:

```
#if HASH_TABLE_SIZE % 2 == 0  HASH_TABLE_SIZE % 3 == 0  \
    HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE shouldn't be divisible by a small prime
#endif
```

The **#warning** command is like the **#error** command, but causes the preprocessor to issue a warning and continue preprocessing. The rest of the line that follows **#warning** is used as the warning message.

You might use **#warning** in obsolete header files, with a message directing the user to the header file which should be used instead.

Pragmas

The **#pragma** command is specified in the ANSI standard to have an arbitrary implementation-defined effect. For example, a **#pragma** might be used to indicate to the translator the best way to generate code, optimize, or diagnose errors. It may also pass information to the translator about the environment, or add debugging information.

The effect of anything specified in a **#pragma** is currently limited to the outermost declaration (that is, a function or a global data declaration).

The following pragmas are implemented in the GNU C Preprocessor:

#pragma CC_OPT_ON	Force optimization on.
#pragma CC_OPT_OFF	Force optimization off.
#pragma CC_OPT_RESTORE	Restore optimization to what was specified on the command line (on if **-O** was specified, off if not).
#pragma CC_WRITABLE_STRINGS	Place strings in the data segment.
#pragma CC_NON_WRITABLE_STRINGS	Place strings in the text segment.

The **#pragma** command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, **#pragma** commands are ignored, except for **#pragma once**.

Combining Source Files

One of the jobs of the C preprocessor is to tell the C compiler the source file and line number that each line of C code came from.

C code can come from multiple source files if you use **#include** or **#import**. If you include header files, or if you use conditionals or macros, the line number of a line in the preprocessor output may be different from the line number of the same line in the original source file. Normally you would want both the C compiler (in error messages) and the GDB debugger to use the line numbers of your source file.

The C preprocessor offers a **#line** command by which you can control this feature explicitly. **#line** specifies the original line number and source file name for subsequent input in the current preprocessor input file. **#line** has three variants:

#line *linenum*
: *linenum* is a decimal integer constant. This resets the current line number in the source file to *linenum.*

#line *linenum* "*file*"
: *linenum* is a decimal integer constant and "file" is a string constant. This resets the line number to *linenum* and changes the name of the file referred to by *file.*

#line *macros*
: *macros* should be one or more macros that have been defined by earlier preprocessing directives. When the macros have been expanded by the preprocessor, the **#line** instruction will then resemble one of the first two forms and be interpreted appropriately.

#line commands alter the results of the **__FILE__** and **__LINE__** predefined macros from that point on. See the section "Predefined Macros."

Miscellaneous Preprocessor Commands

This section describes two additional preprocessor commands. They aren't very useful, but are mentioned for completeness.

- The **null** command consists of a # followed by a newline, with only whitespace (including comments) in between. A null command is understood as a preprocessor command but has no effect on the preprocessor output. The significance of the null command is that an input line consisting of just a # will produce no output, rather than a line of output containing just a #. Some old C programs might contain such lines.

- The **#ident** command is supported for compatibility with certain other systems. It is followed by a line of text. On some systems, the text is copied into a special place in the object file; on most systems, the text is ignored and this directive has no effect. Typically **#ident** is only used in header files supplied with those systems where it is meaningful.

C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessor command lines have been replaced with blank lines and all comments with spaces. White-space characters within a line aren't altered; however, a space is inserted after the expansions of most macros. Also, pragmas are passed through verbatim.

Source file name and line number information is conveyed by lines of the form

# linenum file {digit}

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file *file* at line *linenum. Digit* is 1 if this is the start of a new include file, and 2 if this marks the completion of an include file (this is how the compiler reports the path of inclusion to a given error).

Invoking the C Preprocessor

Usually you won't have to invoke the C preprocessor explicitly, because the C compiler does so automatically. However, there may be times when you want to use the preprocessor by itself by invoking the **cpp** command.

The **cpp** command expects two file names as arguments, *infile* and *outfile.* The preprocessor reads *infile* together with any other files that *infile* specifies by means of **#include** or **#import**. All the output generated by the combined input files is written in *outfile.*

Either *infile* or *outfile* may be **-**, which as *infile* means to read from the standard input and as *outfile* means to write to the standard output. Also, if *outfile* or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here's a list of command options accepted by the C preprocessor. Most of them can also be given when compiling a C program; they're passed along automatically to the preprocessor when it's invoked by the compiler.

-P Inhibit generation of # lines with line-number information in the output from the preprocessor (see the section "C Preprocessor Output"). This might be useful when running the preprocessor on something that isn't C code and that will be sent to a program which might be confused by the # lines.

-C Don't discard comments: Pass them through to the output file. Comments appearing in arguments of a macro invocation will be copied to the output before the expansion of the macro.

-trigraphs Process ANSI standard trigraph sequences.

-pedantic Issue warnings required by the ANSI C standard in certain cases, such as when text other than a comment follows **#else** or **#endif**.

-I*dir* Add the directory *dir* to the end of the list of directories to be searched for header files (see the section "The **#include** Command"). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one **-I** option, the directories are scanned in left-to-right order; the standard system directories come later.

-I- Any directories specified with **-I** options before the **-I-** option are searched only for the case of **#include "***file***"**; they aren't searched for **#include <***file***>**.

If additional directories are specified with **-I** options after the **-I-**, these directories are searched for all **#include** directives.

In addition, the **-I-** option inhibits the use of the current directory as the first search directory for **#include "***file***"**. Therefore, the current directory is searched only if it's requested explicitly with a **-I.** option. Specifying both **-I-** and **-I.** allows you to control precisely which directories are searched before the current one and which are searched after.

-nostdinc Don't search the standard system directories for header files. Only the directories you specify with **-I** options (and the current directory, if appropriate) are searched.

-D*name* Predefine *name* as a macro, with definition **1**.

-D*name=definition*
Predefine *name* as a macro, with definition *definition*.

-U*name* Don't predefine *name*. If both **-U** and **-D** are specified for one name, the name won't be predefined.

-undef Don't predefine any nonstandard macros.

-d Produce a list of **#define** commands for all the macros defined during the execution of the preprocessor, instead of producing the normal preprocessing output.

-M Produce a rule suitable for **make** describing the dependencies of the main source file, instead of outputting the result of preprocessing. The preprocessor produces one **make** rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using backslash-newline.

This feature is used in automatic updating of makefiles.

-MD This is similar to **-M**, but the dependency information is written to files with names made by replacing ".c" with ".d" at the end of the input file names. This is in addition to compiling the file as specified; **-MD** doesn't inhibit ordinary compilation the way **-M** does.

-MM This is similar to **-M**, but mentions only the files included with **#include "***file***"**. System header files included with **#include <***file***>** are omitted.

-MMD This is similar to **-MM**, but mentions only user header files, not system header files.

-H Print the name of each header file used, in addition to other normal activities.

-i*file* Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of **-i***file* is to make the macros defined in *file* available for use in the main input.

13 *The GNU Source-Level Debugger*

13 *The GNU Source-Level Debugger*

This chapter describes how to debug a C program using the GNU debugger from the Free Software Foundation (the GNU debugger has been extended in NeXTSTEP to support the use of Objective C and Mach).

This chapter provides an overview of the GDB debugger and how to use it. The chapter ends with a discussion of NeXTSTEP-specific extensions to GDB. These NeXTSTEP extensions provide full compatibility with standard GDB, while offering the following additional features useful for developing programs within the NeXTSTEP software environment:

- Additional debugger commands
- Extensions to existing debugger commands
- Support for debugging Objective C code

This chapter is a modified version of documentation provided by the Free Software Foundation; see the section "Legal Considerations" at the end of the chapter for important related information.

Summary of GDB

The purpose of a debugger such as GDB is to allow you to execute another program while examining what's going on inside it. We call the other program "your program" or "the program being debugged."

GDB can do four kinds of things (plus other things in support of these):

- Start the program, specifying anything that might affect its behavior.
- Make the program stop on specified conditions.
- Examine what has happened—when the program has stopped—so you can see bugs happen.
- Change things in the program, so you can correct the effects of one bug and go on to learn about another without having to recompile first.

Compiling Your Program for Debugging

To debug a program effectively, you need to ask for debugging information when you compile it. This information in the object file describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the **-g** option when you run the compiler. We recommend that you always use **-g** when you compile a program. You may think the program is correct, but there's no sense in pushing your luck.

Unlike the UNIX C compiler, the GNU C compiler supports debugging with optimization (by using the **-O** compiler option). Although GDB provides the capability to debug programs compiled with optimization, the debugger may provide confusing or misleading information when debugging optimized programs. The intention is to provide some recourse in those situations where debugging optimized programs is necessary. However, debugging optimized programs should not be done routinely.

With these warnings in mind, it can still be useful to debug optimized programs, provided that you're aware of the limitations of the debugger in these circumstances. Most importantly, the debugger should be able to provide correct backtraces of your program's function call stack. This is often all that is needed to find the problem. Printing the values of variables, however, may give incorrect results, since the debugger has insufficient information to be sure where a variable resides at any given time. Variables declared **volatile** will always have correct values, and global variables will almost always be correct; local variables, however, are likely to be incorrectly reported.

Variables declared **register** are optimized by the compiler even when optimizing is not requested with the **-O** compiler option—these may also give misleading results. To ensure a completely predictable debugging environment, it's best to compile without the **-O** flag and with the compiler option "**-Dregister=**". This option causes the C preprocessor to effectively delete all **register** declarations from your program for this compilation. (In fact, with the GNU C compiler, there's no need to declare any variables to be **register** variables. When optimizing, the GNU C compiler may place any variable in a register whether it's declared **register** or not. On the other hand, declaring variables to be **register** variables may make it more difficult to debug your program when not optimizing. Therefore, the use of the **register** declaration is discouraged.)

Running GDB

On a NeXTSTEP computer, you're likely to use GDB by running it within a shell window using a conventional command-line interface—you enter commands at the GDB prompt, and debugger output appears on subsequent lines. (You can also run GDB as a subprocess in the GNU Emacs editor, as described later in this chapter.)

To start GDB from within a shell window, enter the following command:

gdb *name* [*core*]

name is the name of your executable program, and *core*, if specified, is the name of the core dump file to be examined. See the rest of this section for information about optional command-line arguments and switches. Once started, GDB reads commands from the terminal until you quit by giving the **quit** command.

A GDB command is a single line of input. There's no limit to how long it can be. It starts with a command name, optionally followed by arguments (some commands don't allow arguments).

GDB command names may always be abbreviated if the abbreviation is unambiguous. Sometimes even ambiguous abbreviations are allowed. For example, **s** is equivalent to **step** even though there are other commands whose names start with **s**. Possible command abbreviations are stated in the documentation of the individual commands.

A blank line as input to GDB means to repeat the previous command verbatim. Certain commands don't allow themselves to be repeated this way; these are commands for which unintentional repetition might cause trouble and which you're unlikely to want to repeat. Certain others (**list** and **x**) act differently when repeated because that's more useful.

A line of input starting with # is a comment; it does nothing. This is useful mainly in command files (see the section "Command Files").

GDB prompts for commands by displaying the **(gdb)** prompt. You can change the prompt with the **set prompt** command (this is most useful when debugging GDB itself):

set prompt *newprompt*

To exit GDB, use the **quit** command (abbreviated **q**). Control-C won't exit from GDB, but rather will terminate the action of any GDB command that is in progress and return to GDB command level. It's safe to type Control-C at any time because GDB doesn't allow it to take effect until it's safe. If your program is running, typing Control-C will interrupt the program and return you to the GDB prompt.

Specifying Files to Debug

GDB needs to know the file name of the program to be debugged. To debug a core dump of a previous run, GDB must be told the file name of the core dump.

The simplest way to specify the executable and core dump file names is with two command arguments given when you start GDB. The first argument is used as the file for execution and symbols, and the second argument (if any) is used as the core dump file name. Thus,

```
gdb progm core
```

specifies **progm** as the executable program and **core** as a core dump file to examine. (You don't need to have a core dump file if you plan to debug the program interactively.)

If you need to specify more precisely the files to debugged, you can do so with the following command-line options:

-s *file*	Read symbol table from *file*.
-e *file*	Use *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.
-se *file*	Read symbol table from *file* and use it as the executable file.
-c *file*	Use *file* as a core dump file to examine.
-x *file*	Execute GDB commands from *file*.

-d *directory*
Add *directory* to the path to search for source files.

All the options and command line arguments given are processed in sequential order. The order makes a difference when the **-x** command is used.

Specifying GDB Modes

The following additional command-line options can be used to affect certain aspects of the behavior of GDB:

-nx	Don't execute commands from the **.gdbinit** init files. Normally, the commands in these files are executed after all the command options and arguments have been processed. (See the section "Command Files" for more information.)
-q	Quiet. Don't print the usual introductory messages.
-batch	Run in batch mode. Exit with code 1 after processing all the command files specified with **-x** (and **.gdbinit**, if not inhibited). Exit also if, due to an error, GDB would otherwise attempt to read a command from the terminal.
-fullname	This option is used when Emacs runs GDB as a subprocess. It tells GDB to produce the full file name and line number each time a stack frame is displayed (which includes each time the program stops).

Editing GDB Commands

GDB provides a history buffer that stores previously executed commands. You can call any of these commands back to the command line for editing and reexecution. For example, by pressing the up-arrow key repeatedly, you can step back through each of the commands that were issued since the beginning of the session; the down-arrow key steps forward through the history buffer.

Expansion of Variable, Function, and Method Names

GDB supports command-line expansion of variable, function and method names. Type Esc-Esc or Tab to expand the current word on the command line to a matching name. If there is more than one match, the unique part is expanded and a beep occurs. Type Esc-l to display all possible completions.

History Substitution in Commands

GDB supports the **csh** history substitution mechanism. For example, **!foo** retrieves the last command you typed that begins with **foo**. History substitution is supported across **gdb** sessions by writing the command history to a **.gdb_history** file in the current directory. Automatic creation of this history file can be disabled with the command:

```
set history save off
```

History substitution can be contolled with the **set history filename**, **set history size**, **set history save**, and **set history expansion** commands. Also see the section on history substitution in the **csh**(1) UNIX manual page for more information.

Emacs Command-Line Editing

You can use standard Emacs editing commands to edit the contents of the command line. All the basic Emacs command sequences work, as well as the arrow keys. The left and right arrow keys move the cursor along the command line, and the up and down arrow keys take you backward and forward through the command history.

The following list of Emacs commands shows the default key combination associated with each command and a description of what that command does. The name in parentheses can be used to associate a different key combination with the command, as described later in this section.

Insertion-Point Motion Commands

Control-B	Move back one character
Control-F	Move forward one character
Esc b	Move back one word
Esc f	Move forward one word
Control-A	Move to beginning of line
Control-E	Move to end of line

Deletion and Restoration Commands

Control-D	Delete current character
Delete or Control-H	Delete previous character
Esc d	Delete current word
Esc h	Delete previous word
Control-K	Kill forward to end of line
Control-W	Kill region
Control-Y	Restore previous kill from buffer

Search Commands

Control-S	Search forward
Control-R	Search backward
Esc	Exit search mode

History Commands

Esc <	Move to beginning of history file
Esc >	Move to end of history file
Control-N	Go to next history file entry
Control-P	Go to previous history file entry

Miscellaneous Commands

Control-C	Interrupt a program started by the Workspace
Control-L	Clear screen
Control-R	Redisplay current command line
Control-Q	Insert a literal character
Control-I	Insert a Tab
Control-T	Transpose characters
Control-U *n*	Repeat following command *n* times
Control-Z	Suspend debugger, return to shell
Control-@	Set mark

Most of these commands are self-explanatory; the ones requiring more discussion are presented below.

Both delete commands and kill commands erase characters from the command line. Text that's erased by a kill key (Control-K or Control-W) is placed in the "kill buffer." If you want to restore this text, use the "yank" command, Control-Y. The yank command inserts the restored text at the current insertion point. In contrast, text that's erased by one of the delete commands (Control-D, Control-H, Esc d, and Esc h) isn't placed in the kill buffer, so it can't be restored by the yank command.

To enter a character that would otherwise be interpreted as an editing command, you must precede it with Control-Q. For example, to enter Control-D and have it interpreted as a literal rather than as the command to delete the current character, type:

Control-Q Control-D

Editing commands can be repeated by typing Control-U followed by a number and then the command to be repeated. For example, to delete the last 15 characters typed, enter:

Control-U 15 Control-H

If you want to suspend the operation of GDB temporarily and return to the UNIX prompt, type Control-Z. To return to GDB, type **%gdb** (a variant of the shell **fg** command; for more information, see the UNIX manual page for **csh**(1)).

Running GDB in a GNU Emacs Buffer

You can use GNU Emacs to run GDB, as well as to view (and edit) the source files for the program you're debugging with GDB.

To use the Emacs GDB interface, give the command **Esc x gdb** in Emacs. Specify the executable file you want to debug as an argument. This command starts a GDB process as a subprocess of Emacs, with input and output through a newly created Emacs buffer. You can run more than one GDB subprocess by giving the command **Esc x gdb** more than once.

Running GDB as an Emacs subprocess is just like using GDB in a Shell or Terminal window, except for two things:

- All terminal input and output goes through the Emacs buffer. This applies both to GDB commands and their output, and to the input and output done by the program you're debugging. You can copy the text of previous commands and use them again; you can even use parts of the output in this way (all the facilities of Emacs's Shell mode are available for this purpose).
- GDB displays source code through Emacs. Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (=>) at the left margin of the current line.

Explicit GDB list or search commands still produce output as usual, but you'll probably have no reason to use them.

You can use these special Emacs commands in the GDB buffer:

Esc s	Execute to another source line, like the GDB **step** command.
Esc n	Execute to the next source line in this function, skipping all function calls, like the GDB **next** command.

Esc i	Execute one instruction, like the GDB **stepi** command.
Esc u	Move up one stack frame (and display that frame's source file in Emacs), like the GDB **up** command.
Esc d	Move down one stack frame (and display that frame's source file in Emacs), like the GDB **down** command. (You can't use **Esc d** to delete words in the usual fashion in the GDB buffer.)
Control-C Control-F	Execute until exit from the selected stack frame, like the GDB **finish** command.

In any source file, the Emacs command Control-X space (**gdb-break**) tells GDB to set a breakpoint at the source line the point is on.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files in these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of the line numbers as they were at compile time. If you add or delete lines from the text, the line numbers that GDB knows will no longer correspond properly to the code.

Startup Files

At startup, GDB reads configuration information from startup files in the following order:

1. **~/.gdbinit** (your home directory startup file)
2. **./.gdbinit** (the current directory's startup file)

To make your own customizations to GDB, put GDB commands in your home directory's **.gdbinit** startup file. To make further customizations required for any specific project, put commands in a **.gdbinit** startup file within that project's directory.

For more information about making customizations to GDB, see the section "Defining and Executing Sequences of Commands" later in this chapter.

GDB Commands for Specifying and Examining Files

Usually you specify the files for GDB to work with by giving arguments when you invoke GDB. But occasionally it's necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify the files you want to use. In these situations the GDB commands to specify new files are useful.

add-symbol-file *file addr*
: Load the symbols from *file*, assuming *file* has been dynamically loaded. The second argument provides the starting address of the file's text.

core-file [*file*]
: Specify a core dump file to be used as the contents of memory. Note that the core dump contains only the writable parts of memory; the read-only parts must come from the executable file. **core-file** with no argument specifies that no core file is to be used.

  This command has been superseded by the **target core** and **detach** commands.

exec-file [*file*]
: Use *file* as program for getting contents of pure memory. If *file* cannot be found as specified, your execution directory path is searched for a command of that name. No argument means have no executable file.

file [*file*] Use *file* as program to be debugged. It is read for its symbols, for getting the contents of pure memory, and it is the program executed when you use the **run** command. If *file* cannot be found as specified, your execution directory path ($PATH) is searched for a command of that name. No argument means to have no executable file and no symbols.

info files Print the names of the executable and core dump files currently in use by GDB, and the file from which symbols were loaded.

kill Cancel running the program under GDB. This could be used if you want to debug a core dump instead. GDB ignores any core dump file if it's actually running the program, so the **kill** command is the only sure way to go back to using the core dump file.

load *file* Dynamically load *file* into the running program, and record its symbols for access from GDB.

load-file *file*

Dynamically load the specified file into the debugged program. Any symbols are also added to GDB, so you can communicate with the new object file through the use of functions and variables.

path *path*

Add one or more directory to the beginning of the search path for object files. **$cwd** in the path means the current working directory. This path is like the **$PATH** shell variable; it is a list of directories, separated by colons. These directories are searched to find fully linked executable files and separately compiled object files as needed.

symbol-file [*file*]

Read symbol table information from file *file*. The environment variable PATH is searched when necessary. Usually you'll use both the **exec-file** and **symbol-file** commands on the same file.

symbol-file with no argument clears GDB's symbol table.

While file-specifying commands allow both absolute and relative file names as arguments, GDB always converts the file name to an absolute one and remembers it that way.

The **symbol-file** command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

Running Your Program under GDB

To start your program under GDB, use the **run** command. The program must already have been specified using the **exec-file** command or with an argument to the **gdb** command (see the section "Specifying Files to Debug"); what **run** does is create an inferior process, load the program into it, and set it in motion.

The execution of a program is affected by certain types of information it receives from its superior. GDB provides ways to specify these, which you must do before starting the program. (You can change them after starting the program, but such changes don't affect the program unless you start it over again.) The types of information are:

The arguments	You specify the arguments to give the program by passing them as arguments to the **run** command. You can also use the **args** command.
The environment	The program normally inherits its environment from GDB, but you can use the GDB commands **set environment** and **delete environment** to change parts of the environment that will be given to the program.
The working directory	The program inherits its working directory from GDB. You can set GDB's working directory with the **cd** command in GDB.

After the **run** command, the debugger does nothing but wait for your program to stop. See the section "Stopping and Continuing" for more information.

Your Program's Arguments

You specify the arguments to give the program by passing them as arguments to the **run** command. They're first passed to a shell, which expands wildcard characters and performs redirection of I/O, and then passed to the program.

The **run** command with no arguments uses the same arguments used by the previous **run**.

With the **args** command you can specify the arguments to be used the next time the program is run. If **args** has no arguments, it means to use no arguments the next time the program is run. If you've run your program with arguments and want to run it again with no arguments, this is the only way to do so.

Your Program's Environment

Your program's environment consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they're inherited by all the other programs you run. When debugging, it can be useful to try running the program with different environments without having to start the debugger over again.

set environment *varname value*
: Set the environment variable *varname* to *value* (for your program only, not for GDB itself). *value* may be any string; any interpretation is supplied by your program itself.

unset environment *varname*
: Cancel the variable *varname* from the environment passed to your program (thereby making the variable not be defined at all, which is different from giving the variable an empty value). This doesn't affect the program until the next **run** command.

Your Program's Working Directory

Each time you start your program with **run**, the program inherits its working directory from the current working directory of GDB. GDB's working directory is initially whatever it inherited from its superior, but you can specify the working directory for GDB with the **cd** command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See the section "Specifying Files to Debug."

cd *dir* — Set the working directory for GDB and the program being debugged to *dir*. The change doesn't take effect for the program being debugged unitl the next time it is started.

pwd — Print GDB's working directory.

Your Program's Input and Output

By default, the program you run under GDB uses as its source of input and output the same terminal that GDB uses.

You can redirect the program's input and/or output using standard redirection commands with the **run** command. For example,

```
run > outfile
```

starts the program, diverting its output to the file **outfile.**

Another way to specify what the program should use as its source of input and output is with the **tty** command. This command accepts a file name as its argument, and causes that file to be the default for future **run** commands. For example,

```
tty /dev/ttyb
```

causes processes started with subsequent **run** commands to default to using the terminal **/dev/ttyb** as their source of input and output. An explicit redirection in **run** overrides the **tty** command.

When you use the **tty** command or redirect input in the **run** command, the input for your program comes from the specified file, but the input for GDB still comes from your terminal. The program's controlling terminal is your terminal, not the terminal that the program is reading from; so if you want to type Control-C to stop the program, you must type it on your (GDB's) terminal. Control-C typed on the program's terminal is available to the program as ordinary input.

Debugging an Already Running Process

The NeXTSTEP operating system allows GDB to begin debugging an already running process that was started outside GDB. To do this you must use the **attach** command instead of the **run** command.

The **attach** command requires one argument, which is the process ID of the process you want to debug.

attach [*arg*]

Attach to a process or file outside of GDB. This command attaches to another target, of the same type as your last **target** command (**info files** will show your target stack). The command may take as argument a process id or a device file. (The usual way to find out the process ID of the process is with the **ps** utility.) For a process id, you must have permission to send the process a signal, and it must have the same effective uid as the debugger. When using **attach**, you should use the **file** command to specify the program running in the process, and to load its symbol table.

The first thing GDB does after arranging to debug the process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with **run**. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, use the **cont** (continue) command after attaching.

When you're finished debugging the attached process, you can use the **detach** command to detach the debugger from the attached process and resume execution of the process (or you can use Control-C to interrupt the process). After you give the **detach** command, that process and GDB become completely independent, and you're ready to **attach** another process or start one with **run**.

detach

Detach a process or file previously attached. If a process, it is no longer traced, and it continues its execution. If you were debugging a file, the file is closed and GDB no longer accesses it.

If you exit GDB or use the **run** command while you have an attached process, you kill that process. You'll be asked for confirmation if you try to do either of these things.

The following commands are for connecting to a target machine or process.

target [*args*]

Connect to a target machine or process. The first argument is the type or protocol of the target machine. Remaining arguments are interpreted by the target protocol. For more information on the arguments for a particular protocol, type **help target** followed by the protocol name.

target child

Unix child process (started by the **run** command).

target core *file*

Use a core file as a target. Specify the filename of the core file.

target exec *file*

Use an executable file as a target. Specify the filename of the executable file.

target remote *device*

Use a remote computer via a serial line, using a GDB-specific protocol. Specify the serial device it is connected to (for example, **/dev/ttya**).

The following commands are for kernel debugging.

kattach *hostname*

Attach to a kernel on a remote host.

kreboot *args*

Reboot an attached kernel.

Stopping and Continuing

When you run a program normally, it runs until exiting. The purpose of using a debugger is so that you can stop it before that point, or so that if the program runs into trouble you can find out why.

Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, SIGINT is the signal a program gets when you type Control-C; SIGSEGV is the signal a program gets from referencing a place in memory far away from all the areas in use; SIGALRM occurs when the alarm clock timer goes off (which happens only if the program has requested an alarm).

Some signals, including SIGALRM, are a normal part of the functioning of the program. Others, such as SIGSEGV, indicate errors; these signals are fatal (that is, they kill the program immediately) if the program hasn't specified in advance some other way to handle the signal. SIGINT doesn't indicate an error in the program, but it's normally fatal, so it can carry out the purpose of Control-C: to kill the program.

GDB can detect any occurrence of a signal in the program running under GDB's control. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like SIGALRM (so as not to interfere with their role in the functioning of the program) but to stop the program immediately whenever an error signal happens. You can change these settings with the **handle** command. You must specify which signal you're talking about with its number.

info signals [*signalnum*]
: Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals. Specify a signal number in order to print information about that signal only.

handle *signalnum keywords*
: Change the way GDB handles signal *signalnum.* The *keywords* say what change to make.

To use the **handle** command you must know the code number of the signal you're concerned with. To find the code number, type **info signal**; this prints a table of signal names and numbers.

The keywords allowed by the **handle** command can be abbreviated. Their full names are:

stop	GDB should stop the program when this signal happens. This implies the **print** keyword as well.
print	GDB should print a message when this signal happens.
nostop	GDB shouldn't stop the program when this signal happens. It may still print a message telling you that the signal has come in.
noprint	GDB shouldn't mention the occurrence of the signal at all. This implies the **nostop** keyword as well.
pass	GDB should allow the program to see this signal; the program will be able to handle the signal, or may be terminated if the signal is fatal and not handled.
nopass	GDB shouldn't allow the program to see this signal.

When a signal has been set to stop the program, the program can't see the signal until you continue. It will see the signal then, if **pass** is in effect for the signal in question at that time. In other words, after GDB reports a signal, you can use the **handle** command with **pass** or **nopass** to control whether that signal will be seen by the program when you later continue it.

You can also use the **signal** command to prevent the program from seeing a signal, to cause it to see a signal it normally wouldn't see, or to give it any signal at any time. See the section "Continuing" below.

Breakpoints

A breakpoint can be used to make your program stop whenever a certain point in the program is reached. You set breakpoints explicitly with GDB commands, specifying the place where the program should stop by line number, function name, or exact address in the program. You can add various other conditions to control whether the program will stop.

Each breakpoint is assigned a number when it's created; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints, you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be "enabled" or "disabled;" if disabled, it has no effect on the program until you enable it again.

The **info breakpoints** command prints a list of all breakpoints set and not cleared, showing their numbers, their location in the program, and any special features in use for them. Disabled breakpoints are included in the list, but marked as disabled. **info breakpoints** with a breakpoint number as its argument lists only that breakpoint. The convenience variable $_ and the default address for the **x** command are set to the address of the last breakpoint listed (see the section "Examining Memory"). The **info breakpoints** command can be abbreviated as **info break.**

Breakpoints can't be used in a program if any other process is running that program. Attempting to run or continue the program with a breakpoint in this case will cause GDB to stop it. When this happens, you have two ways to proceed:

- Remove or disable the breakpoints, then continue.
- Suspend GDB, and copy the file containing the program to a new name. Resume GDB and use the **exec-file** command to specify that GDB should run the program under that name. Then start the program again.

Setting Breakpoints

Breakpoints are set with the **break** command (abbreviated **b**). There are several ways to specify where the breakpoint should go:

break *function*
: Set a breakpoint at entry to *function*. You can also set a breakpoint at the entry to a method, as described in the section "Method Names in Commands."

break *linenum*
: Set a breakpoint at *linenum* in the current source file (the last file whose source text was printed). This breakpoint will stop the program just before it executes any of the code from that line.

break *file*:*linenum*
: Set a breakpoint at *linenum* in *file*.

break *file*:*function*
: Set a breakpoint at entry to *function* found in *file*. Specifying a file name as well as a function name is superfluous except when multiple files contain identically named functions.

break **address*
: Set a breakpoint at *address*. You can use this to set breakpoints in parts of the program that don't have debugging information or source files.

break
: Set a breakpoint at the next instruction to be executed in the selected stack frame (see the section "Examining the Stack"). This is a pointless thing to do in the innermost stack frame because the program would stop immediately after being started, but it's very useful with another stack frame, because it will cause the program to stop as soon as control returns to that frame.

break [*args*] **if** *cond*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero. *args* stands for one of the possible arguments described above (or no argument) specifying where to break. See the section "Break Conditions" for more information.

tbreak [*args*]
: Set a breakpoint enabled only for one stop. *args* are the same as in the **break** command, and the breakpoint is set in the same way, but the breakpoint is automatically "disabled" the first time it's hit.

GDB allows you to set any number of breakpoints at the same place in the program. This can be useful when the breakpoints are conditional (see the section "Break Conditions").

Clearing Breakpoints

It's often necessary to eliminate a breakpoint once it has done its job and you no longer want the program to stop there. This is called clearing (or deleting) the breakpoint. A breakpoint that has been cleared no longer exists in any sense.

With the **clear** command you can clear breakpoints according to where they are in the program. With the **delete** command you can clear individual breakpoints by specifying their breakpoint numbers.

It isn't necessary to clear a breakpoint to proceed past it. GDB automatically ignores breakpoints in the first instruction to be executed when you continue execution at the same address where the program stopped.

clear
: Clear any breakpoints at the next instruction to be executed in the selected stack frame (see the section "Selecting a Frame"). When the innermost frame is selected, this is a good way to clear a breakpoint that the program just stopped at.

clear *function*
: **clear** *file***:***function*
: Clear any breakpoints set at entry to the *function.*

clear *linenum*
: **clear** *file***:***linenum*
: Clear any breakpoints set at or within the code of the specified line.

delete *bnum ...*
: Clear the breakpoints whose breakpoint numbers are specified as arguments. A deleted breakpoint is forgotten completely.

Disabling Breakpoints

Rather than clearing a breakpoint, you might prefer to disable it. This makes the breakpoint inoperative as if it had been cleared, but remembers the information about the breakpoint so that you can enable it again later.

You enable and disable breakpoints with the **enable** and **disable** commands, specifying one or more breakpoint numbers as arguments. Use **info breakpoints** to print a list of breakpoints if you don't know which breakpoint numbers to use.

A breakpoint can have any of four states of enablement:

- Disabled. The breakpoint has no effect on the program.
- Enabled. The breakpoint will stop the program. A breakpoint made with the **break** command starts out in this state.
- Enabled once. The breakpoint will stop the program, but when it does so it will become disabled. A breakpoint made with the **tbreak** command starts out in this state.
- Enabled for deletion. The breakpoint will stop the program, but immediately afterward it will be deleted permanently.

You can change the state of enablement of a breakpoint with the following commands:

enable *bnum ...*
: Enable the specified breakpoints. They become effective once again in stopping the program, until you specify otherwise.

enable once *bnum ...*
: Enable the specified breakpoints temporarily. Each will remain enabled only until the next time it stops the program (unless you use one of these commands to specify a different state before that time comes). Also see the **tbreak** command, which sets a breakpoint and enables it once.

enable delete *bnum ...*
: Enable the specified breakpoints to work once and then die. Each of the breakpoints will be deleted the next time it stops the program (unless you use one of these commands to specify a different state before that time comes).

disable delete *bnum ...*
: Disable the specified breakpoints. A disabled breakpoint has no effect but isn't forgotten. All options such as ignore counts, conditions, and commands are remembered in case the breakpoint is enabled again later. This command may be abbreviated **disable**.

delete breakpoints [*bnum ...*]
: Delete some breakpoints or auto-display expressions. Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument. This command may be abbreviated **delete**.

enable display [*arg* ...]

Enable some expressions to be displayed when the program stops. Arguments are the code numbers of the expressions to resume displaying. No argument means enable all automatic-display expressions. Do **info display** to see the current list of code numbers.

disable display [*arg* ...]

Disable some expressions to be displayed when the program stops. Arguments are the code numbers of the expressions to stop displaying. No argument means disable all automatic-display expressions. Do **info display** to see the current list of code numbers.

delete display [*arg* ...]

Cancel some expressions to be displayed when the program stops. Arguments are the code numbers of the expressions to stop displaying. No argument means cancel all automatic-display expressions. Do **info display** to see the current list of code numbers.

catch [*arg* ...]

Set breakpoints to catch exceptions that are raised. Argument may be a single exception to catch, multiple exceptions to catch, or the default exception **default**. If no arguments are given, breakpoints are set at all exception handlers catch clauses within the current scope.

A condition specified for the catch applies to all breakpoints set with this command.

Aside from the automatic disablement or deletion of a breakpoint when it stops the program, which happens only in certain states, the state of enablement of a breakpoint changes only when one of the above commands is used.

Break Conditions

The simplest sort of breakpoint breaks every time the program reaches a specified place. You can also specify a condition for a breakpoint. A condition is simply a boolean expression. A breakpoint with a condition evaluates the expression each time the program reaches it, and the program stops only if the condition is true.

Break conditions may have side effects, and may even call functions in your program. These may sound like strange things to do, but their effects are completely predictable unless there's another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop the program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached (see the section “Executing Commands at a Breakpoint”).

Break conditions can be specified when a breakpoint is set, by using **if** in the arguments to the **break** command (see the section “Setting Breakpoints”). They can also be changed at any time with the **condition** command:

condition *bnum expression*

Specify *expression* as the break condition for breakpoint number *bnum*. From now on, this breakpoint will stop the program only if the value of *expression* is true (nonzero, in C). *expression* isn’t evaluated at the time the **condition** command is given.

condition *bnum*

Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

A special feature is provided for one kind of condition: to prevent the breakpoint from doing anything until it has been reached a certain number of times. This is done with the “ignore count” of the breakpoint. When the program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by 1 and continues.

ignore *bnum count*

Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, it won’t stop.

To make the breakpoint stop the next time it’s reached, specify a count of 0.

continue *n*

Continue execution of the program, setting the ignore count of the breakpoint that the program stopped at to *n* minus 1. Continuing through the breakpoint doesn’t itself count as one of *n*. Thus, the program won’t stop at this breakpoint until the *n*th time it’s hit.

This command is allowed only when the program stopped due to a breakpoint. At other times, the argument to **cont** is ignored.

If a breakpoint has a positive ignore count and a condition, the condition isn’t checked. Once the ignore count reaches 0, the condition will start to be checked.

You could achieve the effect of the ignore count with a condition such as **$foo--<= 0** using a debugger convenience variable that’s decremented each time. That’s why the ignore count is considered a special case of a condition. See the section “Convenience Variables.”

Executing Commands at a Breakpoint

You can give any breakpoint a series of commands to execute when the program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

commands *bnum*

Specify commands for breakpoint number *bnum*. The commands themselves appear on the following lines. Type a line containing just **end** to terminate the commands.

To remove all commands from a breakpoint, use the command **commands** and follow it immediately by **end**; that is, give no commands.

Breakpoint commands can be used to start up the program again. Simply use the **continue** command, or **step**, or any other command that resumes execution. However, any remaining breakpoint commands are ignored. When the program stops again, GDB will act according to why that stop took place.

If the first command specified is **silent**, the usual message about stopping at a breakpoint isn't printed. This may be desirable for breakpoints that are to print a specific message and then continue. If the remaining commands also print nothing, you'll see no sign that the breakpoint was reached at all. **silent** isn't really a command; it's meaningful only at the beginning of the commands for a breakpoint.

The commands **echo** and **output**, which allow you to print precisely controlled output, are often useful in silent breakpoints. See the section "Commands for Controlled Output."

Here's how you could use breakpoint commands to print the value of **x** at entry to **foo** whenever it's positive. We assume that the newly created breakpoint is number 4; **break** will print the number that's assigned.

```
break foo if x>0
commands 4
silent
echo x is\040
output x
echo \n
cont
end
```

One application for breakpoint commands is to correct one bug so you can test another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the **cont** command so that the program doesn't stop, and start with the **silent** command so that no output is produced. Here's an example:

```
break 403
commands 5
silent
set x = y + 4
cont
end
```

One deficiency in the operation of breakpoints that continue automatically appears when your program uses raw mode for the terminal. GDB reverts to its own terminal modes (not raw) before executing commands, and then must switch back to raw mode when your program is continued. This causes any pending terminal input to be lost.

You could get around this problem by putting the actions in the breakpoint condition instead of in commands. For example,

```
condition 5  (x = y + 4), 0
```

is a condition expression that will change **x** as needed, then always have the value 0 so the program won't stop. Loss of input is avoided here because break conditions are evaluated without changing the terminal modes. When you want to have nontrivial conditions for performing the side effects, the operators **&&**, **||** , and **?:** may be useful.

Continuing

After your program stops, most likely you'll want it to run some more if the bug you're looking for hasn't happened yet. You can do this with the **continue** command:

continue Continue running the program at the place where it stopped.

If the program stopped at a breakpoint, the place to continue running is the address of the breakpoint. You might expect that continuing would just stop at the same breakpoint immediately. In fact, **continue** takes special care to prevent that from happening. You don't need to clear the breakpoint to proceed through it after stopping at it.

You can, however, specify an ignore count for the breakpoint that the program stopped at, by means of an argument to the **continue** command. See the section "Break Conditions" above.

If the program stopped because of a signal other than SIGINT or SIGTRAP, continuing will cause the program to see that signal. You may not want this to happen. For example, if the program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but the program would probably terminate immediately as a result of the fatal signal once it sees the signal. To prevent this, you can continue with **signal 0**. You can also act in advance to prevent the program from seeing certain kinds of signals, using the **handle** command (see the section "Signals").

Stepping

Stepping means setting your program in motion for a limited time, so that control will return automatically to the debugger after one line of code or one machine instruction. Breakpoints are active during stepping and the program will stop for them even if it hasn't gone as far as the stepping command specifies.

step [*count*]

Proceed the program until control reaches a different line, then stop it and return to the debugger. If an argument is specified, proceed as in **step**, but do so *count* times. If a breakpoint or a signal not related to stepping is reached before *count* steps, stepping stops right away. You can abbreviate this command as **s**.

next [*count*]

Similar to **step**, but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the **next** command was given. An argument is a repeat count, as in **step**. You can abbreviate this command as **n**.

finish

Continue running until just after the selected stack frame returns (or until there's some other reason to stop, such as a fatal signal or a breakpoint). Upon return, the value returned is printed and put in the value history. Contrast this with the **return** command, described in the section "Returning from a Function."

until *linenum*

Continue running until line number *linenum* is reached or the current stack frame returns. This is equivalent to setting a breakpoint at *linenum*, executing a **finish** command, and deleting the breakpoint.

stepi [*count*]

Proceed one machine instruction, then stop and return to the debugger. It's often useful to do **display/i $pc** when stepping by machine instructions. This will cause the next instruction to be executed to be displayed automatically at each stop (see the section "Automatic Display"). An argument is a repeat count, as in **step**. You can abbreviate this command as **si**.

nexti [*count*]

Proceed one machine instruction, but if it's a subroutine call, proceed until the subroutine returns. An argument is a repeat count, as in **next**. You can abbreviate this command as **ni**.

watch *exp*

Single-step the program until *exp* is true. This method is slow, but accurate. For a similar method that's faster but less accurate, see the section on data breakpoints.

A typical technique for using stepping is to put a breakpoint at the beginning of the function or the section of the program in which a problem is believed to lie, and then step through the suspect area examining interesting variables until the problem happens.

The **cont** command can be used after stepping to resume execution until the next breakpoint or signal.

Data Breakpoints

Data breakpoints are implemented using a scheme that involves calling a handler function at the end of every function. This allows the program to break at the end of the function that changed the data, thereby narrowing the search for the offending line to a space between the last function called within a function and the last line of that function. The offending line is somewhere before the stopping point—specifically, somewhere between the end of the last function that was called and the end of this function.

The following commands provide support for data breakpoints:

data-break [*address size*]
data-break [*expression*]

Causes the program to break at the end of the function that changes the specified data. The first form specifies that the data starts at *address* for *size* bytes. The second form specifies that the data starts at *&expression* and continue for the size of *expression* (an Objective C object is considered to be the size that its class dictates at the time of the **data-break** command). With no arguments, **data-break** removes any **data-break** condition currently in effect.

set-exit-handler [*function-name*]

Causes *function-name* to be called every time a function is exited. The prototype of the function is int (handlerFunction)(void). If a non-zero value is returned, the program will break at the end of the last function that was called. With no arguments, this command removes the exit handler.

There are two caveats. First, functions without frame pointers are exempt from checking. Second, the address being checked must be readable at all times that the data breakpoint handler will be called. Otherwise an exception will be generated inside the inferior program. **gdb** will catch this, and you'll have to turn off checking by using the **data-break** command with no arguments.

The following examples illustrate the use of the **data-break** command.

- The program will stop if anything in the range 0x1000 through 0x100b changes:

  ```
  data-break 0x1000 12
  ```

- An assigment to **foo** will cause the program to stop:

  ```
  char *foo;
  data-break foo
  ```

- An assigment to **foo[0]** will cause the program to stop:

  ```
  char *foo;
  foo = malloc(20);
  data-break *foo
  ```

- An assigment to any of the characters from **foo[0]** through **foo[19]** will cause the program to stop:

  ```
  char *foo;
  data-break foo 20
  ```

- An assigment to **foo** will cause the program to stop:

```
int foo;
data-break foo
```

- An assignment to **foo** will cause the program to stop.

```
struct _foo {
    int a;
    int b;
};
struct _foo *foo;
data-break foo
```

- An assignment to **foo->a** will cause the program to stop:

```
struct _foo {
    int a;
    int b;
};
struct _foo *foo;
foo = malloc(sizeof(*foo));
data-break foo->a
```

- An assignment to **foo->a** or **foo->b** will cause the program to stop:

```
struct _foo {
    int a;
    int b;
};
struct _foo *foo;
foo = malloc(sizeof(*foo));
data-break *foo
```

- An assignment to **foo** will cause the program to stop:

```
id foo;
data-break foo
```

- An assigment to **foo->appSpeaker** will cause the program to stop:

```
id foo = [Application new];
data-break foo->appSpeaker
```

- An assignment to any of the instance variables of **foo** will cause the program to stop:

```
id foo;
data-break *foo
```

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, the information about where in the program the call was made from is saved in a block of data called a *stack frame*. The frame also contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of memory called the call stack. When your program stops, the GDB commands for examining the stack allow you to see all this information.

Stack Frames

The call stack is divided into contiguous pieces called frames; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function **main()**. This is called the initial frame, or the outermost frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the innermost frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing the address of one of those bytes to serve as the address of the frame. Usually this address is kept in a register called the frame pointer register while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with 0 for the innermost frame, 1 for the frame that called it, and so on upward. These numbers don't really exist in your program; they simply give you a way of talking about stack frames in GDB commands.

At any given time, one of the stack frames is selected by GDB; many GDB commands refer implicitly to this selected frame. In particular, whenever you ask GDB for the value of a variable in the program, the value is found in the selected frame. You can select any frame using the **frame**, **up**, and **down** commands; subsequent commands will operate on that frame.

When the program stops, GDB automatically selects the currently executing frame and describes it briefly, as the **frame** command does (see the section "Information about a Frame").

Backtraces

A backtrace is a summary of how the program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame 0) followed by its caller (frame 1), and on up the stack.

Each line in a backtrace shows the frame number, the program counter, the function and its arguments, and the source file name and line number (if known). For example:

```
(gdb) backtrace
#0  0x3eb6 in fflush ()
#1  0x24b0 in _fwalk ()
#2  0x2500 in _cleanup ()
#3  0x2312 in exit ()
```

backtrace [*n*]

Print a backtrace of the entire stack: one line per frame for all frames in the stack. You can stop the backtrace at any time by typing the system interrupt character, normally Control-C. With a positive argument, the command prints the innermost *n* frames; with a negative argument, it prints the outermost *n* frames. You can abbreviate this command as **bt**. An alias for this command is **where**.

Selecting a Frame

Most commands for examining the stack and other data in the program work on whichever stack frame is selected at the moment. Below are the commands for selecting a stack frame.

frame *n*

Select and print frame number *n*. Recall that frame 0 is the innermost (currently executing) frame, frame 1 is the frame that called the innermost one, and so on. The highest-numbered frame is **main**'s frame.

frame *addr*

Select and print the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful if the program has multiple stacks and switches between them.

up *n* Select and print the frame *n* frames up from the frame previously selected. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to 1.

up-silently *n*
Same as the **up** command, but doesn't print anything (this is useful in command scripts).

down *n* Select and print the frame *n* frames down from the frame previously selected. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to 1.

down-silently *n*
Same as the **down** command, but doesn't print anything (this is useful in command scripts).

All these commands (except **up-silently** and **down-silently**) end by printing some information about the frame that has been selected: the frame number, the function name, the arguments, the source file and line number of execution in that frame, and the text of that source line. For example:

```
#3  main (argc=3, argv=??, env=??) at main.c, line 67
67      read_input_file (argv[i]);
```

After such a printout, the **list** command with no arguments will print ten lines centered on the point of execution in the frame. See the section "Printing Source Lines."

Information about a Frame

There are several other commands to print information about the selected stack frame.

frame [*n*] This command prints a brief description of the selected stack frame. With an argument, this command is used to select a stack frame (the argument can be a stack frame number or the address of a frame); with no argument, it doesn't change which frame is selected, but still prints the same information. You can abbreviate this command as **f**.

info frame

This command prints a verbose description of the selected stack frame, including the address of the frame, the addresses of the next frame down (called by this frame) and the next frame up (caller of this frame), the address of the frame's arguments, the program counter saved in it (the address of execution in the caller frame), and which registers were saved in the frame. The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *addr*

Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command.

info args Print the arguments of the selected frame, each on a separate line.

info locals Print the local variables of the selected frame, each on a separate line.

Examining Source Files

GDB knows which source files your program was compiled from, and can print parts of their text. When your program stops, GDB spontaneously prints the line it stopped in. Likewise, when you select a stack frame (see the section "Selecting a Frame"), GDB prints the line in which execution in that frame has stopped. You can also print parts of source files by explicit command.

Viewing Files in Edit

To be able to dynamically open and view source files in Edit, use the **view** command. This command is executed automatically when you start GDB from the Project Builder application.

view [*host*]

Cause source files to be viewed in Edit, either on the local machine or on a remote *host*. When you first execute this command, a Gdb command is added to Edit's main menu. This command brings up a control panel that allows you to control some of GDB's basic functions from within Edit (see Chapter 4 for details).

unview Cause source files not to be viewed in Edit.

Printing Source Lines

To print lines from a source file, use the **list** command (abbreviated **l**). There are several ways to specify what part of the file you want to print.

Here are the most commonly used forms of **the list** command:

list *linenum*
: Print ten lines centered around *linenum* in the current source file.

list *function*
: Print ten lines centered around the beginning of *function.*

list
: Print ten more lines. If the last lines printed were printed with a **list** command, this prints ten lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see the section "Examining the Stack"), this prints ten lines centered around that line.

list –
: Print ten lines just before the lines last printed.

You can repeat a **list** command by pressing the Return key; however, any argument that was used is discarded, so this is equivalent to typing simply **list**. An exception is made for an argument of **-**; that argument is preserved in repetition so that each repetition moves up in the file.

In general, the **list** command expects you to supply zero, one, or two linespecs. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. The possible arguments for **list** are as follows:

list *,last*
: Print ten lines ending with *last.*

list *first***,**
: Print ten lines starting with *first.*

list +
: Print ten lines just after the lines last printed.

list –
: Print ten lines just before the lines last printed.

list *linespec*
: Print ten lines centered around the line specified by *linespec* (described below).

list *first,last*
: Print lines from *first* to *last.* Both arguments are *linespec*s.

Here are the possible ways to specify a value for *linespec*:

linenum Specifies line *linenum* of the current source file. When a **list** command has two *linespec*s, this refers to the same source file as the first *linespec*.

+offset Specifies the line *offset* lines after the last line printed. When used as the second *linespec* in a **list** command, this specifies the line *offset* lines down from the first *linespec*.

–offset Specifies the line *offset* lines before the last line printed.

file:*linenum*
Specifies line *linenum* in the source file *file*.

function Specifies the line of the left brace ({) that begins the body of *function*.

file:*function*
Specifies the line of the left brace ({) that begins the body of *function* in *file*. The file name is needed with a function name only for disambiguating identically named functions in different source files.

**addr* Specifies the line containing the program address *addr*. *addr* may be any expression.

The **info line** command is used to map source lines to program addresses:

info line [*line*]
Print the starting and ending addresses of the compiled code for source line *line*, which can be specified as:

linenum, to list around that line in current file,
file:*linenum*, to list around that line in that file,
function, to list around beginning of that function, or
file:*function*, to distinguish among like-named static functions.

With no argument, the command describes the last source line that was listed.

The default address for the **x** command is changed to the starting address of the line, so that **x/i** is sufficient to begin examining the machine code (see the section "Examining Memory"). Also, this address is saved as the value of the convenience variable **$_** (see the section "Convenience Variables").

Searching Source Files

The **forward-search** command (or its alias, **search**) and the **reverse-search** command are useful when you want to locate text within the current source file.

forward-search *regexp*
: This command checks each line, starting with the one following the last line listed, for a match for *regexp*, which must be a UNIX regular expression (see the UNIX manual page for **ed**). It lists the line that's found. You can abbreviate this command as **fo**.

reverse-search *regexp*
: The command checks each line, starting with the one before the last line listed and going backward, for a match for *regexp*. It lists the line that's found. You can abbreviate this command as **rev**.

Specifying Source Directories

Executable programs don't record the directories of the source files they were compiled from, just the names. GDB remembers a list of pathnames of directories in which it will search for source files; this list is called the source path (note that GDB doesn't use the environment variable PATH to search for source files). Each time GDB wants a source file, it tries each directory in the list, starting from the beginning, until it finds a file with the desired name.

When you start GDB, its source path is set to **$cdir:$cwd** (the current working directory, and the directory in which the source file was compiled into object code). To add other directories, use the **directory** command:

directory *dirname*
: Add directory with the pathname *dirname* to the beginning of the source path.

directory Reset the source path to **$cdir:$cwd**, the default.This requires confirmation.

Examining Data

The most common way to examine data in your program is with the **print** command (abbreviated **p**):

print *exp* This command evaluates and prints the value of any valid expression of the language the program is written in (currently, only C and Objective C). Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

exp is any valid expression, and the value of *exp* is printed in a format appropriate to its data type. To print data in another format, you can cast *exp* to the desired type or use the **x** command.

$*num* gets previous value number *num*. $ and $$ are the last two values. $$*num* refers to the *num*'th value back from the last one. Names starting with $ refer to registers (with the values they would have if the program were to return to the stack frame now selected, restoring all registers saved by frames farther in) or else to debugger convenience variables (any such name that isn't a known register). Use assignment expressions to give values to convenience variables.

{*type*}*adrexp* refers to a datum of data type *type*, located at address *adrexp*. @ is a binary operator for treating consecutive data objects anywhere in memory as an array. *foo*@*num* gives an array whose first element is *foo*, whose second element is stored in the space following where *foo* is stored, etc. *foo* must be an expression whose value resides in memory.

exp may be preceded with /*fmt*, where *fmt* is a format letter but no count or size letter (see the description of the **x** command).

print-object *object*
Print *object* by sending **printForDebugger:** to it.

set *exp* The **set** command works like the **print** command, except that the expression's value isn't displayed. This is useful for modifying the state of your program. For example:

```
set x=3
set close_all_files()
```

Another way to examine data is with the **x** command (see "Examining Memory" below). It examines data in memory at a specified address and prints it in a specified format.

Expressions

Many different GDB commands accept an expression and compute its value. Any kind of constant, variable, or operator defined by the programming language you're using is legal in an expression in GDB. This includes conditional expressions, function calls, casts, and string constants.

GDB supports three kinds of operator in addition to those of programming languages:

file-or-function::*variable-name*
: :: allows you to specify a variable in terms of the file or function it's defined in.

@
: @ is a binary operator for treating parts of memory as arrays. See the section "Artificial Arrays" below for more information.

{*type*} *addr*
: Refers to an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around nonunary operators, just as in a cast). This construct is allowed no matter what kind of data is officially supposed to reside at *addr*.

Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see the section "Selecting a Frame"); they must be either global (or static) or visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

```
foo (a)
    int a;
{
    bar (a);
    {
        int b = test ();
        bar (b);
    }
}
```

the variable **a** is usable whenever the program is executing within the function **foo()**, but the variable **b** is usable only while the program is executing inside the block in which **b** is declared.

Artificial Arrays

It's often useful to print out several successive objects of the same type in memory (for example, a section of an array, or an array of dynamically determined size for which only a pointer exists in the program).

This can be done by constructing an "artificial array" with the binary operator @. The left operand of @ should be the first element of the desired array, as an individual object. The right operand should be the length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. For example, if a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of **array** with

```
p *array@len
```

The left operand of @ must reside in memory. Array values made with @ in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions.

Output Formats

GDB normally prints all values according to their data types. Sometimes this isn't what you want. For example, you might want to print a number in hexadecimal, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or an instruction. These things can be done with output formats.

The simplest use of output formats is to specify how to print a value already computed. This is done by starting the arguments of the **print** command with a slash and a format letter. The format letters supported are:

x	Regard the bits of the value as an integer, and print the integer in hexadecimal.
d	Print as integer in signed decimal.
u	Print as integer in unsigned decimal.
o	Print as integer in octal.
a	Print as an address, both absolute in hexadecimal and then relative to a symbol defined at an address below it.

c	Regard as an integer and print as a character constant.
f	Regard the bits of the value as a floating-point number and print using typical floating-point syntax.

For example, to print the program counter in hexadecimal (see the section “Registers”), type

```
p/x $pc
```

No space is required before the slash because command names in GDB can’t contain a slash.

To reprint the last value in the value history with a different format, you can use the **print** command with just a format and no expression. For example, **p/x** reprints the last value in hexadecimal.

Examining Memory

The command **x** (for “examine”) can be used to examine memory under explicit control of formats, without reference to the program’s data types.

x is followed by a slash and an output format specification, followed by an expression for an address:

x/*fmt addr*

The expression *addr* doesn’t need to have a pointer value (though it may); it’s used as an integer, as the address of a byte of memory.

The output format *fmt* in this case specifies both how big a unit of memory to examine and how to print the contents of that unit. It’s done with one or two of the letters listed below.

These letters specify the size of unit to examine:

b	Examine individual bytes.
h	Examine halfwords (two bytes each).
w	Examine words (four bytes each).
g	Examine giant words (eight bytes).

These letters specify how to print the contents:

x	Print as integers in unsigned hexadecimal.
d	Print as integers in signed decimal.
u	Print as integers in unsigned decimal.
o	Print as integers in unsigned octal.
a	Print as an address, both absolute in hexadecimal and then relative to a symbol defined as an address below it.
c	Print as character constants (this implies size **b**).
f	Print as floating point. This works only with sizes **w** and **g**.
s	Print a null-terminated string of characters. The specified unit size is ignored; instead, the unit is however many bytes it takes to reach a null character (including the null character).
i	Print a machine instruction in assembler syntax (or nearly). The specified unit size is ignored; the number of bytes in an instruction varies depending on the type of machine, the opcode and the addressing modes used.

If neither the manner of printing nor the size of unit is specified, the default is the same as was used last. If you don't want to use any letters after the slash, you can omit the slash as well.

You can also omit the address to examine. Then the address used is just after the last unit examined. This is why string and instruction formats actually compute a unit-size based on the data: so that the next string or instruction examined will start in the right place. The **print** command sometimes sets the default address for the **x** command; when the value printed resides in memory, the default is set to examine the same location. **info line** also sets the default for **x**, to the address of the start of the machine code for the specified line and **info breakpoints** sets it to the address of the last breakpoint listed.

When you repeat an **x** command by pressing the Return key, the address specified previously (if any) is ignored; instead, the command examines successive locations in memory rather than the same one.

You can examine several consecutive units of memory with one command by writing a repeat count after the slash (before the format letters, if any). The repeat count must be a decimal integer. It has the same effect as repeating the **x** command that many times except that the output may be more compact with several units per line.

```
x/10i $pc
```

Prints ten instructions starting with the one to be executed next in the selected frame. After doing this, you could print another ten following instructions with

```
x/10
```

in which the format and address are allowed to default.

The addresses and contents printed by the **x** command aren't put in the value history because there's often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables **$_** and **$__** (that is, **$** followed by one or two underscores).

After an **x** command, the last address examined is available for use in expressions in the convenience variable **$_**. The contents of that address, as examined, are available in the convenience variable **$__**.

If the **x** command has a repeat count, the address and contents saved are from the last memory unit printed; this isn't the same as the last address printed if several units were printed on the last line of output.

Automatic Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the "automatic display list" so that GDB will print its value each time the program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

showing item numbers, expressions, and their current values.

display *exp*

Add the expression *exp* to the list of expressions to display each time the program stops.

display/*fmt exp*

Add the expression *exp* to the automatic display list, and display it in the format *fmt*. *fmt* should specify only a display format, not a size or count.

display/*fmt addr*

Add the expression *addr* as a memory address to be examined each time the program stops. *fmt* should be either **i** or **s**, or it should include a unit size or a number of units. See the section "Examining Memory."

undisplay [*n* ...]

Remove item number *n* from the list of expressions to display. With no argument, cancels all automatic-display expressions.

display

Display the current values of the expressions on the list, just as is done when the program stops.

info display

Print the list of expressions to display automatically, each one with its item number, but without showing the values.

Value History

Every value printed by the **print** command is saved for the entire session in GDB's "value history" so that you can refer to it in other expressions.

The values printed are given "history numbers" for you to refer to them by. These are successive integers starting with 1. **print** shows you the history number assigned to a value by printing $\$n$ = before the value, where *n* is the history number.

To refer to any previous value, use **$** followed by the value's history number. The output printed by **print** is designed to remind you of this. **$** alone refers to the most recent value in the history, and **$$** refers to the value before that.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It's enough to type

```
p *$
```

If you have a chain of structures where the component **next** points to the next one, you can print the contents of the next one with

```
p *$.next
```

It might be useful to repeat this command many times by pressing the Return key.

Note that the history records values, not expressions. If the value of **x** is 4 and you type

```
print x
set x=5
```

then the value recorded in the value history by the **print** command remains 4 even though **x**'s value has changed.

Convenience Variables

GDB provides "convenience variables" that you can use within GDB to hold a value for future reference. These variables exist entirely within GDB; they aren't part of your program, and setting a convenience variable has no effect on further execution of your program. That's why you can use them freely.

Convenience variables have names starting with **$**. Any name starting with **$** can be used for a convenience variable, unless it's one of the predefined set of register names (see the section "Registers").

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. For example:

```
set $foo = *object_ptr
```

would save in **$foo** the value contained in the object pointed to by **object_ptr**.

Convenience variables don't need to be explicitly declared; using a convenience variable for the first time creates it. However, its value is **void** until you assign it a value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, even if it already has a value of a different type. The convenience variable as an expression has whatever type its current value has.

One way to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example:

```
set $i = 0
print bar[$i++]->contents
    repeat that command by typing RET.
```

Some convenience variables are created automatically by GDB and given values likely to be useful.

$_
: The variable **$_** (single underscore) is automatically set by the **x** command to the last address examined (see the section "Examining Memory"). Other commands which provide a default address for **x** to examine also set **$_** to that address; these commands include **info line** and **info breakpoint**.

$__
: The variable **$__** (two underscores) is automatically set by the **x** command to the value found in the last address examined.

Registers

Machine register contents can be referred to in expressions as variables with names starting with $.

The names **$pc** and **$sp** are used for the program counter register and the stack pointer. **$fp** is used for a register that contains a pointer to the current stack frame. To see a list of all the registers, use the command **info registers**.

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system isn't the same one that your program normally sees. For example, the registers of the 68882 floating-point coprocessor are always saved in "extended" format, but all C programs expect to work with "double" format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the **info registers** command prints the data in both formats.

Register values are relative to the selected stack frame (see the section "Selecting a Frame"). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the real contents of all registers, you must select the innermost frame (with **frame 0**).

Some registers are never saved (typically those numbered 0 or 1) because they're used for returning function values; for these registers, relativization makes no difference.

info registers [*regname*]
: With no argument, print the names and relativized values of all registers. With an argument, print the relativized value of register *regname*. *regname* may be any register name valid on the machine you're using, with or without the initial $.

For example, you could print the program counter in hexadecimal with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add 4 to the stack pointer with

```
set $sp += 4
```

The last is a way of removing one word from the stack. This assumes that the innermost stack frame is selected. Setting **$sp** isn't allowed when other stack frames are selected.

Miscellaneous Commands

call *arg*
: Call a function in the inferior process. The argument is the function name and arguments, in standard C notation. The result is printed and saved in the value history, if it isn't void.

disassemble [*arg* [*arg*]]
: Disassemble a specified section of memory. The default is the function surrounding the **pc** of the selected frame. With a single argument, the function surrounding that address is dumped. Two arguments are taken as a range of memory to dump.

whereis *variable*
: Print the location of the specified variable.

Examining the Symbol Table

The commands described in this section allow you to make inquiries for information about the symbols (names of variables, functions, and types) defined in your program. GDB finds this information in the symbol table contained in the executable file; it's inherent in the text of your program and doesn't change as the program executes.

whatis [*exp*]

With no argument, print the data type of **$**, the last value in the value history. With an argument, print the data type of expression *exp*. *exp* isn't actually evaluated, and any operations inside it that have side effects (such as assignments or function calls) don't take place.

info address *symbol*

Describe where the data for *symbol* is stored. For register variables, this says which register. For other automatic variables, this prints the stack-frame offset at which the variable is always stored. Note the contrast with **print &***symbol*, which doesn't work at all for register variables, and which for automatic variables prints the exact address of the current instantiation of the variable.

info functions [*regexp*]

With no argument, print the names and data types of all defined functions. With an argument, print the names and data types of all defined functions whose names contain a match for regular expression *regexp* (for information about regular expressions, see the UNIX manual page for **ed**). For example, **info fun step** finds all functions whose names include **step; info fun ^step** finds those whose names start with **step**.

info sources

Print the names of all source files in the program for which there is debugging information.

info types [*regexp*]

With no argument, print all data types that are defined in the program. With an argument, print all data types that are defined in the program whose names contain a match for regular expression *regexp*.

info variables [*regexp*]

With no argument, print the names and data types of all top-level variables that are declared outside functions. With an argument, print the names and data types of all variables declared outside functions, whose names contain a match for regular expression *regexp*.

printsyms *file*

Write a complete dump of the debugger's symbol data into the file *file*.

ptype *typename*

Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form **struct** *struct-tag*, **union** *union-tag* or **enum** *enum-tag*. The selected stack frame's lexical context is used to look up the name.

Setting Variables

set

Perform an assignment *var* = *exp*. You must type the =. *var* may be a debugger convenience variable (a name starting with $), a register (one of a few standard names starting with $), or an actual variable in the program being debugged. *exp* is any expression. Use **set variable** for variables with names identical to **set** subcommands.

With a subcommand listed below, the **set** command modifies parts of the GDB environment (you can see these environment settings with **show** and its subcommands). In general, use **on** (or no argument) to enable a feature, and **off** to disable it.

set args *arg* ...

Set arguments to give the program being debugged when it is started. Follow this command with any number of arguments to be passed to the program.

set autoload-breakpoints *on/off*

Set automatic resetting of breakpoints in dynamic code.

set autoload-symbols *on/off*

Set automatic loading of symbols of dynamic code.

set catch-user-commands-errors *on/off*

Set whether to ignore errors in user commands.

set complaints *num*

Set the maximum number of complaints about incorrect symbols.

set confirm *on/off*

Set whether to confirm potentially dangerous operations.

set editing *on/off*

Set command-line editing.

set environment *var value*

Set environment variable and value to give the program. Arguments are *var value* where *var* is the variable name and *value* is the value. Values of environment variables are uninterpreted strings. This command does not affect the program until the next **run** command.

set history expansion *on/off*

Set history expansion on command input.

set history filename *file*

Set the filename in which to record the command history (the list of previous commands of which a record is kept).

set history save *on/off*

Set whether the history record is saved when you exit **gdb**.

set history size *size*

Set the size of the command history (the number of previous commands to keep a record of).

set lazy-read *on/off*

Set whether inferior's memory is read lazily.

set print address *on/off*

Set printing of addresses.

set print array *on/off*

Set pretty printing of arrays.

set print asm-demangle *on/off*

Set demangling of C++ names in disassembly listings.

set print demangle *on/off*

Set demangling of encoded C++ names when displaying symbols.

set print elements *size*

Set limit on string chars or array elements to print. The value **0** causes there to be no limit.

set print object *on/off*

Set printing of object's derived type based on vtable info.

set print sevenbit-strings *on/off*

Set printing of 8-bit characters in strings as *nnn*.

set print vtbl *on/off*

Set printing of C++ virtual function tables.

set print union *on/off*

Set printing of unions interior to structures.

set print pretty *on/off*

Set pretty printing of structures.

set prompt *string*

Set GDB's prompt. The argument is an unquoted string.

set radix *on/off*

Set the default input and output number radix.

set unload-symbols *on/off*

Set whether symbols from dynamically loaded code are forgotten for future loads.

set verbose *on/off*

Set whether verbose printing of informational messages is enabled or disabled.

set view-host *host*

Set the host to connect to when viewing.

set view-program *name*

Set the name of the program to connect to when viewing.

set variable *var* = *exp*

Same as **set**; use **set variable** in cases where *var* is identical to one of the **set** subcommands.

Status Inquiries

info address *var*

Describe where the specified variable is stored.

info args Provide information about the argument variables of the current stack frame.

info breakpoints [*num*]

Provide information about the status of all breakpoints, or of breakpoint number *num*. The second column displays **y** for enabled breakpoints, **n** for disabled, **o** for enabled once (disable when hit), or **d** for enabled but delete when hit. The address and the file/line number are also displayed.

The convenience variable **$_** and the default examine address for **x** are set to the address of the last breakpoint listed. The convenience variable **$bpnum** contains the number of the last breakpoint set.

info catch

Provide information about the exceptions that can be caught in the current stack frame.

info classes

Show all Objective C classes.

info copying

Show conditions for redistributing copies of GDB.

info display

Show expressions to display when program stops, with code numbers.

info files Show the names of targets and files being debugged. Shows the entire stack of targets currently in use (including the exec-file, core-file, and process, if any), as well as the symbol file name.

info float Show the status of the floating point unit.

info frame [*addr*]

Provide information about the selected stack frame, or the frame at *addr.*

info functions [*regexp*]

Show all function names, or those matching *regexp*.

info line [*line_spec*]

Core addresses of the code for a source line. *line_spec* can be specified as

linenum, to list around that line in current file,
*file***:***linenum*, to list around that line in that file,
function, to list around beginning of that function, or
*file***:***function*, to distinguish among like-named static functions.

The default is to describe the last source line that was listed.

This sets the default address for **x** to the line's first instruction so that **x/i** suffices to start examining the machine code. The address is also stored as the value of **$_**.

info locals

Provide information about the local variables of the current stack frame.

info program

Show the execution status of the program.

info registers [*register_name*]

Show a list of registers and their contents for the selected stack frame. A register name as argument means describe only that register.

info selectors
Show all Objective C selectors.

info set Show all GDB settings.

info signals [*sig_num*]
Show what GDB does when the program gets various signals. Specify a signal number to print information about that signal only.

info sources
Show the names of source files in the program.

info source
Provide information about the current source file.

info stack [*count*]
Provide a backtrace of the stack, or of the innermost *count* frames.

info target
Same as **info files**.

info terminal
Print inferior's saved terminal status.

info types [*regexp*]
Show all type names, or those matching *regexp*.

info user [*command*]
Show the definition of a user-defined command. With no argument, show the definitions of all user-defined commands.

info variables [*regexp*]
Show all global and static variable names, or those matching *regexp*.

info warranty
Show information pertaining to warranty.

info watchpoints [*num*]
Provide information about the status of all watchpoints, or of watchpoint number *num*. The second column displays **y** for enabled watchpoints or **n** for disabled ones.

show autoload-breakpoints
Show automatic reseting of breakpoints in dynamic code.

show autoload-symbols
Show automatic loading of symbols of dynamic code.

show args Show arguments to give program being debugged when it is started.

show catch-user-commands-errors
Show whether to ignore errors in user commands.

show commands
Show the status of the command editor.

show complaints
Show the maximum number of complaints about incorrect symbols.

show confirm
Show whether to confirm potentially dangerous operations.

show convenience
Show the debugger convenience variables. These variables are created when you assign them values; thus, **print $foo=1** gives **$foo** the value 1. Values may be of any type.

A few convenience variables are given values automatically: **$_** holds the last address examined with **x** or **info lines**, and **$__** holds the contents of the last address examined with **x**.

show directories
Current search path for finding source files. **$cwd** in the path means the current working directory. **$cdir** in the path means the compilation directory of the source file.

show editing
Show command-line editing.

show environment [*var*]
Show the environment to give the program, or one variable's value. With an argument *var*, prints the value of environment variable *var* to give the program being debugged. With no arguments, prints the entire environment to be given to the program.

show history expansion
Show history expansion on command input.

show history filename

Show the filename in which to record the command history (the list of previous commands of which a record is kept).

show history save

Show saving of the history record on exit.

show history size

Show the size of the command history (that is, the number of previous commands to keep a record of).

show lazy-read

Show if inferior's memory is read lazily.

show paths

Show the current search path for finding object files. **$cwd** in the path means the current working directory. This path is like the **$PATH** shell variable; that is, a list of directories separated by colons. These directories are searched to find fully linked executable files and separately compiled object files as needed.

show print address

Show printing of addresses.

show print array

Show prettyprinting of arrays.

show print asm-demangle

Show demangling of C++ names in disassembly listings.

show print demangle

Show demangling of encoded C++ names when displaying symbols.

show print elements

Show limit on string chars or array elements to print.

show print object

Show printing of object's derived type based on vtable info.

show print pretty

Show pretty printing of structures.

show print sevenbit-strings

Show printing of 8-bit characters in strings as \nnn.

show print union
Show printing of unions interior to structures.

show print vtbl
Show printing of C++ virtual function tables.

show prompt
Show GDB's prompt.

show radix
Show the default input and output number radix.

show unload-symbols
Show whether symbols from dynamically loaded code are forgotten for future loads.

show values [*idx*]
Elements of value history around item number *idx* (or last ten).

show verbose
Show whether verbosity is on or off.

show version
Report what version of GDB this is.

show view-host
Show host to connect to when viewing.

show view-program
Show name of program to connect to when viewing.

Debugging PostScript Code

This section describes three commands that are useful when debugging PostScript source files.

These commands aren't built-in commands; rather, the NeXTSTEP environment defines them in a system **.gdbinit** file located in the directory **/usr/lib**. This file is read when you start running GDB (the contents of this file are shown later in this chapter).

showps **shownops**The **showps** and **shownops** commands turn on and off (respectively) the display of PostScript code being sent from your application to the Window Server. Your application must be running before you can issue either of these commands.

flush The **flush** command sends pending PostScript code to the Window Server. This command lets you flush the application's output buffer, causing any PostScript code waiting there to be interpreted immediately. Your application must be running before you can issue this command.

Debugging Objective C Code

This section provides information about some commands and command options that are useful for debugging Objective C code.

Method Names in Commands

The following commands have been extended to accept Objective C method names as line specifications:

clear
break
info line
jump
list

For example, to set a breakpoint at the **create** instance method of class Fruit in the program currently being debugged, enter:

```
break [Fruit create]
```

It's also possible to specify just a method name:

```
break create
```

If your program's source files contain more than one **create** method, you'll be presented with a numbered list of classes that implement that method. Indicate your choice by number, or type 0 to exit if none apply. To narrow the scope of GDB's search, you can use a preceding plus or minus sign to specify whether you're referring to a class or an instance method. For example, to list the ten program lines around the initialize class method, enter

```
list +[Text initialize]
```

or

```
list +initialize
```

You must specify the complete method name, including any colons. For example, to clear a breakpoint established at the **orderWindow:relativeTo:** method of the Window class, enter:

```
clear [Window orderWindow:relativeTo:]
```

Command Descriptions

This section describes commands and options that are useful in debugging Objective C code. Some of these are new commands that have been implemented in NeXTSTEP, and some are previously existing GDB commands that have been extended in NeXTSTEP.

The info Command

The **info** command takes three additional options:

info classes [*regexp*]
Display all Objective C classes in your application, or those matching the regular expression *regexp*.

info selectors [*regexp*]
Display all Objective C selector names (or those matching the regular expression *regexp*), and also each selector's unique number.

If you don't limit the command's scope by entering a regular expression, the resulting listing can be quite long. To terminate a listing at any point and return to the GDB prompt, type Control-C.

Two standard **info** command options have been extended. The **info types** command recognizes and lists the Objective C **id** type. The **info line** command recognizes Objective C method names as line specifications.

The print Command

The **print** command has been extended to allow the evaluation of Objective C objects and message expressions. Consider, for example, this program excerpt:

```
@implementation Fruit : Object
{
    char *color;
    int diameter;
}

+ create  {
    id newInstance;
    newInstance = [super new];        // creates instance of Fruit
    [ newInstance color:"green" ];    // set the color
    [ newInstance diameter:1];        // set the diameter
    return newInstance;               // return the new instance
}
. . .
@end
```

Once this code has been executed, you can use GDB to examine **newInstance** by entering:

```
print newInstance
```

The output looks something like this (of course, the address wouldn't be the same):

```
$1 = (id) 0x1a020
```

As declared, **newInstance** is a pointer to an Objective C object. To see the structure this variable points to, enter:

```
print *newInstance
```

GDB displays:

```
$3 = {
    isa = 0x120b4;
    color = 0x26bf "green";
    diameter = 1;
}
```

This structure contains the instance variables defined above for objects of the Fruit class. It also contains a pointer, called **isa**, that points to its class object. To see the identity of this class, enter:

```
print *newInstance->isa
```

GDB displays:

```
$4 = {
    isa = 0x12090;
    super_class = 0x124a4;
    name = 0x125a2 "Fruit";
    version = 0;
    info = 17;
    instance_size = 12;
    ivars = 0x1203c;
    methods = 0x120ec;
    cache = 0x22080;
}
```

The instance variable **name** verifies that this is an instance of the Fruit class.

You can also evaluate a message expression with the **print** command. As a by-product of the evaluation, the message is sent to the receiving object. For example, the following command sets the color of the Fruit object to red:

```
print [newInstance color: "red"]
```

The set Command

The **set** command can be used to evaluate and send a message expression. For example, the following command sets the color of the Fruit object to red:

```
set [newInstance color: "red"]
```

The step Command

The **step** command has been extended to let you step through the execution of an Objective C message. By repeatedly executing the **step** command, you can watch the chain of events that make up the execution of a message.

If you step into a message and don't want to follow the details of its execution, enter:

```
finish
```

This command completes the execution of the message and stops the program at the next statement. To avoid stepping into the message in the first place, use the **next** command rather than **step**. The **next** command instructs GDB to execute the current command and stop only when control returns to the current stack frame.

Debugging Mach Threads

The following commands have been provided in the NeXTSTEP version of GDB to support the debugging of Mach threads.

thread-list *thread*
  List all threads that exist in the program being debugged (abbreviated **tl**).

thread-select *thread*
  Select a thread (abbreviated **ts**). For example, **ts 2** selects thread 2.

tsuspend *thread*
  Suspend execution of *thread.*

tresume *thread*
  Resumes execution of a particular thread.

Debugging NeXTSTEP Core Files

NeXTSTEP GDB has been extended to allow debugging of NeXTSTEP core files, which are in the Mach-O file format. Core files are generated in the **/cores** directory, if it exists; otherwise, they're generated in the current working directory.

The **info files** command lists information about the contents of the core file. This tells you what segments of address space exist in the core file, how many threads exist in the core image, and what the program counter is for each thread. Thread 0 is selected by default, so if you do a **bt** it will apply to thread 0. The **thread-list** and **thread-select** commands, documented in the section "Debugging Mach Threads" above, work with core files. All the normal debugger commands can also be used while debugging the core image.

Altering Execution

There are several ways to alter the execution of your program with GDB commands.

Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. For example:

```
print x=4
```

would store the value 4 into the variable **x**, and then print the value of the assignment expression (which is 4).

If you aren't interested in seeing the value of the assignment, use the **set** command instead of the **print** command. **set** is the same as **print** except that the expression's value isn't printed and isn't put in the value history. The expression is evaluated only for side effects.

GDB allows more implicit conversions in assignments than C does; you can freely store an integer value into a pointer variable or vice versa, and any structure can be converted to any other structure that's the same length or shorter.

All the other C assignment operators such as **+=** and **++** are supported as well.

To store into arbitrary places in memory, use the **{...}** construct to generate a value of specified type at a specified address. For example:

```
set {int}0x83040 = 4
```

Continuing at a Different Address

jump *linenum*

Resume execution at line number *linenum*. Execution may stop immediately if there's a breakpoint there.

The **jump** command doesn't change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If *linenum* is in a different function from the one currently executing, the results may be wild if the two functions expect different patterns of arguments or of local variables. For this reason, the **jump** command requests confirmation if the specified line isn't in the function currently executing.

jump **address*

Resume execution at the instruction at address *address*.

A somewhat similar effect can be obtained by storing a new value into the register **$pc**. For example:

```
set $pc = 0x485
```

specifies the address at which execution will resume, but doesn't resume execution. That doesn't happen until you use the **cont** command or a stepping command.

Returning from a Function

return [*exp*]

You can make any function call return immediately by using the **return** command.

First select the stack frame that you want to return from (see the section "Selecting a Frame"). Then type the **return** command. If you want to specify the value to be returned, give that as an argument.

The selected stack frame (and any other frames inside it) is popped, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The **return** command doesn't resume execution; it leaves the program stopped in the state that would exist if the function had just returned. Contrast this with the **finish** command, which resumes execution until the selected stack frame returns naturally.

Defining and Executing Sequences of Commands

GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

User-Defined Commands

A "user-defined command" is a sequence of GDB commands to which you assign a new name as a command. This is done with the **define** command.

define *commandname*

Define a command named *commandname*. If there's already a command by that name, you're asked to confirm that you want to redefine it.

The definition of the command is made up of other GDB command lines, which are given following the **define** command. The end of the command definition is marked by a line containing just the command **end**. For example:

```
define w
    where
end
```

document *commandname*

Create documentation for the user-defined command *commandname*. The command *commandname* must already be defined. This command reads lines of documentation just as **define** reads the lines of the command definition. After the **document** command is finished, **help** on command *commandname* will print the documentation you have specified.

You may use the **document** command again to change the documentation of a command. Redefining the command with **define** doesn't change the documentation, so be sure to keep the documentation up to date.

User-defined commands don't take arguments. When they're executed, the commands of the definition aren't printed. An error in any command stops execution of the user-defined command.

Commands that would ask for confirmation if used interactively proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they're doing omit the messages when used in a user-defined command.

Command Files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting with #) may also be included. An empty line in a command file does nothing; it doesn't cause the last command to be repeated, as it would from the terminal.

When GDB starts, it automatically executes its "init files" (command files named **.gdbinit**). GDB first reads the init file (if any) in your home directory and then the init file (if any) in the current working directory. (The init files aren't executed if the **-nx** option is given.) You can also request the execution of a command file with the **source** command:

source *file*

Execute the command file *file*.

The lines in a command file are executed sequentially. They aren't printed as they're executed. An error in any command terminates execution of the command file.

Commands that would ask for confirmation if used interactively proceed without asking when used in a command file. Many GDB commands that normally print messages to say what they're doing omit the messages when used in a command file.

Commands for Controlled Output

During the execution of a command file or a user-defined command, the only output that appears is what's explicitly printed by the commands of the definition. This section describes three additional commands useful for generating exactly the output you want.

echo *text* Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as **\n** to print a newline. No newline will be printed unless you specify one.

A backslash at the end of *text* is ignored. It's useful for producing a string ending in spaces, since trailing spaces are trimmed from all arguments. A backslash at the beginning preserves leading spaces in the same way, because the escape sequence backslash-space stands for a space. Thus, to print " variable foo = ", do

```
echo \ variable foo = \
```

output *expression*

Print just the value of *expression*. A newline character isn't printed, and the value isn't entered in the value history.

output/*fmt expression*

Print the value of *expression* in format *fmt*. See the section "Debugging PostScript Code" for more information.

printf *format-string, arg* [, *arg*] ...

Print the values of the arguments, under the control of *format-string*. This command is identical in its operation to its C library equivalent (see the UNIX manual page for **printf()** for format codes).

Miscellaneous Commands

browse [*object*]

Browse an object

dump-me

Produce a fatal error and make GDB dump its core.

dump-strings [*file*]

Dump all the strings seen into *file*.

make [*args*]

Run the **make** program using the rest of the line as arguments.

select-frame

Select the frame at **fp, pc**.

shell [*command*]

Execute the rest of the line as a shell command. With no arguments, run an inferior shell.

Legal Considerations

Permission is granted to make and distribute verbatim copies of this chapter provided its copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this chapter under the conditions for verbatim copying, provided also that the section entitled "GDB General Public License" (below) is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this chapter into another language, under the above conditions for modified versions, except that the section entitled "GDB General Public License" may be included in a translation approved by the author instead of in the original English.

Distribution

GNU software is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. GNU software is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of GNU software that they might get from you. The precise conditions are found in the GNU General Public License that appears following this section.

You may obtain a complete machine-readable copy of any NeXTSTEP-modified source code for Free Software Foundation software under the terms of Free Software foundation's general public licenses, without charge except for the cost of media, shipping and handling, upon written request to Technical Services at NeXT Computer, Inc.

When making a request, please specify which GNU software programs you're interested in receiving. GNU programs released by NeXT currently include:

gcc	GNU compiler
gdb	GNU debugger
gas	GNU assembler
emacs	GNU text editor

If you want an unmodified, verbatim copy of any GNU software (including GNU software that's not part of the NeXTSTEP software release), you can order it from the Free Software Foundation. Though GNU software itself is free, the distribution service is not. For further information, write to:

Free Software Foundation
675 Mass. Ave.
Cambridge, MA 02139

Income that Free Software Foundation derives from distribution fees goes to support the Foundation's purpose: the development of more free software to distribute.

GDB General Public License

The license agreements of most software companies keep you at the mercy of those companies. By contrast, our general public license is intended to give everyone the right to share GDB. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

Specifically, we want to make sure that you have the right to give away copies of GDB, that you receive source code or else can get it if you want it, that you can change GDB or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of GDB, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for GDB. If GDB is modified by someone else and passed on, we want its recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

Therefore we (Richard Stallman and the Free Software Foundation, Inc.) make the following terms which say what you must do to be allowed to distribute or change GDB.

Copying Policies

1. You may copy and distribute verbatim copies of GDB source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy a valid copyright notice "Copyright (c) 1988 Free Software Foundation, Inc." (or with whatever year is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of the GDB program a copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.

2. You may modify your copy or copies of GDB or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

- cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of GDB or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).

- You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another unrelated program with this program (or its derivative) on a volume of a storage or distribution medium does not bring the other program under the scope of these terms.

3. You may copy and distribute GDB (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,

   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal shipping charge) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs.

4. You may not copy, sublicense, distribute or transfer GDB except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GDB is void and your rights to use the program under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.

No Warranty

14 *Mach Object Files*

14 *Mach Object Files*

This chapter describes the format of Mach object files. This format is used by default, rather than the UNIX 4.3BSD **a.out** format, for object files on NeXTSTEP computers.

The current Mach object format is still evolving at Carnegie Mellon, and enhancements in NeXTSTEP are part of this evolving process. These enhancements refine the design and clean up some implementation details. The concepts of the original design are still present, but names have been changed for consistency.

The Mach object file format has two components:

- A static header containing information common to all files

- A variable number of load commands that provide information about the structure of the file

The load commands provide the following types of information:

- The layout of the run-time memory image
- The symbol table information
- The initial thread execution state
- The names of any referenced shared libraries

The layout of the file is determined by the file type:

- For types MH_EXECUTE and MH_FVMLIB the segments are padded out and aligned on a segment alignment boundary for efficient demand paging. Both these file types also have the headers included as part of their first segment.

- The type MH_OBJECT is a compact format (the ".o" format). It's intended only as output of the assembler and input (or possibly output) of the link editor. All sections are in one unnamed segment with no padding.

- The type MH_PRELOAD is an executable format intended for files that aren't executed under the kernel (such as PROMs, standalone programs, and kernels).
- The type MH_CORE is for core files.

The structures of a Mach object file are defined in the header file **mach-o/loader.h**, and are described below. The structures and what they're used for are described first, followed by a list of what structures make up Mach object files.

The Mach Header

The Mach header appears at the beginning of the object file. Only information that's truly general to the file is contained in the Mach header. Other information is put in the load commands that follow.

The format of the Mach header is:

```
struct mach_header {
    unsigned long  magic;      /* Mach magic number identifier */
    cpu_type_t     cputype;    /* cpu specifier */
    cpu_subtype_t  cpusubtype; /* machine specifier */
    unsigned long  filetype;   /* type of file */
    unsigned long  ncmds;      /* number of load commands */
    unsigned long  sizeofcmds; /* size of all load commands */
    unsigned long  flags;      /* flags */
};
```

The value for the **magic** field of the **mach_header** structure is:

```
#define MH_MAGIC    0xfeedface  /* the Mach magic number */
```

The values for the **cputype** and **cpusubtype** fields are defined as follows in the header file **sys/machine.h**:

```
#define CPU_TYPE_MC680x0     ((cpu_type_t) 6)
#define CPU_SUBTYPE_MC68030  ((cpu_subtype_t) 1)
#define CPU_SUBTYPE_MC68040  ((cpu_subtype_t) 2)
```

The values for the **filetype** field are defined as follows in the header file **sys/loader.h**:

```
#define MH_OBJECT   0x1    /* relocatable object file */
#define MH_EXECUTE  0x2    /* executable object file */
#define MH_FVMLIB   0x3    /* fixed vm shared library file */
#define MH_CORE     0x4    /* core file */
#define MH_PRELOAD  0x5    /* preloaded executable file */
```

The **ncmds** field contains the number of **load_command** structures that follow the Mach header. The **load_command** structures directly follow the Mach header in the object file.

The **sizeofcmds** field contains the total size in bytes of all of the load commands that follow it.

The following constants are used for the **flags** field:

```
#define MH_NOUNDEFS  0x1  /* object file has no undefined references;
                             can be executed */
#define MH_INCRLINK  0x2  /* object file is the output of an
                             incremental link against a base file;
                             can't be link-edited again */
```

The Load Commands

The load commands appear directly after the Mach header. They are variable in size. The number of load commands and the total size of the commands are given in the **ncmds** and **sizeofcmds** fields of the **mach_header** structure.

All load commands must have as their first two fields **cmd** and **cmdsize**:

- The **cmd** field contains a constant for that command type. Each command type has a specific structure corresponding to it.
- The **cmdsize** field is the size in bytes of the particular **load_command** structure plus anything that follows it that's a part of the load command (for example, **section** structures or strings). To advance to the next load command, the value of the **cmdsize** field can be added to the offset or pointer of the current load command.

The value of the **cmdsize** field must be a multiple of **sizeof(long)**. This is the maximum alignment of any load command. The padded bytes must be zero-filled. Because the file will be memory mapped, all tables in the object file must also follow these rules; otherwise the pointers to these tables are not guaranteed to work. With all padding zero-filled, like objects will compare byte for byte.

The following structure is the minimum form of a load command:

```
struct load_command {
    unsigned long  cmd;       /* type of load command */
    unsigned long  cmdsize;   /* total size of command in bytes */
};
```

Constants for the **cmd** field of the **load_command** structure are:

```
#define LC_SEGMENT     0x1   /* file segment to be mapped */
#define LC_SYMTAB      0x2   /* link-edit stab symbol table info
                                (obsolete) */
#define LC_SYMSEG      0x3   /* link-edit gdb symbol table info */
#define LC_THREAD      0x4   /* thread */
#define LC_UNIXTHREAD  0x5   /* UNIX thread (includes a stack) */
#define LC_LOADFVMLIB  0x6   /* load a fixed VM shared library */
#define LC_IDFVMLIB    0x7   /* fixed VM shared library id */
#define LC_IDENT       0x8   /* object identification information
                                (obsolete) */
#define LC_FVMFILE     0x9   /* fixed VM file inclusion */
```

A variable-length string in a load command is represented by an **lc_str** union. The string is stored just after the **load_command** structure, and the offset is from the start of the **load_command** structure. The size of the string is reflected in the **cmdsize** field of the load command. Any padded bytes to bring the **cmdsize** field to a multiple of **sizeof(long)** must be zero-filled.

```
union lc_str {
    unsigned long  offset;    /* offset to the string */
    char          *ptr;       /* pointer to the string */
};
```

The LC_SEGMENT Load Command

The LC_SEGMENT load command indicates that a part of this file is to be mapped into the task's address space. The size of this segment in memory, **vmsize**, can be equal to or larger than the amount to map from this file, **filesize**. The file, starting at **fileoff**, is mapped to the beginning of the segment in memory at **vmaddr**. The rest of the memory of the segment, if any, is allocated zero-fill on demand.

```
struct segment_command {
    unsigned long  cmd;           /* LC_SEGMENT */
    unsigned long  cmdsize;       /* includes size of section
                                     structures */
    char           segname[16];   /* segment's name */
    unsigned long  vmaddr;        /* segment's memory address */
    unsigned long  vmsize;        /* segment's memory size */
    unsigned long  fileoff;       /* segment's file offset */
    unsigned long  filesize;      /* amount to map from file */
    vm_prot_t      maxprot;       /* maximum VM protection */
    vm_prot_t      initprot;      /* initial VM protection */
    unsigned long  nsects;        /* number of sections */
    unsigned long  flags;         /* flags */
};
```

The segment's maximum virtual memory protection and initial virtual memory protection are specified by the **maxprot** and **initprot** fields. The values for these fields are set to some combination of the constants defined in the header file **vm/vm_prot.h**:

```
#define VM_PROT_NONE    ((vm_prot_t) 0x00)
#define VM_PROT_READ    ((vm_prot_t) 0x01)  /* read permission */
#define VM_PROT_WRITE   ((vm_prot_t) 0x02)  /* write permission */
#define VM_PROT_EXECUTE ((vm_prot_t) 0x04)  /* execute permission */

/* The default protection for newly created virtual memory */
#define VM_PROT_DEFAULT  \
        (VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE)

/* Maximum privileges possible, for parameter checking. */
#define VM_PROT_ALL  \
        (VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE)
```

A segment's address and virtual memory protection are set at link edit time.

The following constants can be used for the **flags** field of the **segment_command** structure:

```
#define SG_HIGHVM   0x1
#define SG_FVMLIB   0x2
#define SG_NORELOC  0x3
```

SG_HIGHVM indicates that the file contents for this segment occupy the high part of the virtual memory space; the low part is zero-filled (for stacks in core files). SG_FVMLIB indicates that the segment is the virtual memory that's allocated by a fixed virtual memory library for overlap checking in the link editor. SG_NORELOC indicates that the segment has nothing that was relocated in it and nothing relocated to it (that is, it may be safely replaced without relocation).

A segment is made up of zero or more sections. If the segment contains sections, the section structures directly follow the segment command and their size is reflected in the **cmdsize** field.

If sections have the same section name and are going into the same segment, they're combined by the link editor. The resulting section is aligned to the maximum alignment of the combined sections and is the new section's alignment. The combined sections are aligned to their original alignment in the combined section. Any padded bytes used to get the specified alignment are zero-filled.

Only non-MH_OBJECT files have all their segments with the proper sections in each padded to the specified segment alignment. The default segment alignment for the link editor is the page size. The first segment of an executable or shared library always contains the Mach header and load commands of the object file before its first section. The zero-filled sections are always last in their segment, allowing the zeroed segment padding to be mapped into memory where zero-filled sections might be.

```
struct section {
    char  sectname[16];        /* section's name */
    char  segname[16];         /* segment the section is in */
    unsigned long  addr;       /* section's memory address */
    unsigned long  size;       /* section's size in bytes */
    unsigned long  offset;     /* section's file offset */
    unsigned long  align;      /* section's alignment */
    unsigned long  reloff;     /* file offset of relocation entries */
    unsigned long  nreloc;     /* number of relocation entries */
    unsigned long  flags;      /* flags */
    unsigned long  reserved1;  /* reserved */
    unsigned long  reserved2;  /* reserved */
};
```

Flags currently defined for the **flags** field of a **section** structure are the following:

```
#define S_ZEROFILL         0x1  /* zero-filled on demand */
#define S_CSTRING_LITERALS 0x2  /* section has only literal C
                                   strings */
#define S_4BYTE_LITERALS   0x2  /* section has only 4-byte literals */
#define S_8BYTE_LITERALS   0x2  /* section has only 8-byte literals */
#define S_LITERAL_POINTERS 0x2  /* section has only pointers to
                                   literals */
```

S_ZEROFILL is used for the uninitialized data sections; sections with literal flags cause the link editor to coalesce redundant literals into sections and perform the proper relocation, resulting in a smaller file.

The format of the relocation entries referenced by the **reloff** and **nreloc** fields is described in the header file **reloc.h.**

Although the names of segments and sections in them are mostly meaningless to the link editor, there are a few things to support traditional UNIX executables that will require the link editor and assembler to use some agreed-upon names.

The link editor will allocate common symbols at the end of the **__common** section in the **__DATA** segment, creating the section and segment if needed. The **__common** section must be a zero-fill section (marked with S_ZEROFILL).

The default **maxprot** and **initprot** (maximum and initial virtual memory protection) will always be read, write, and execute. If there's a **__TEXT** or **__LINKEDIT** segment its **initprot** won't be writable by default.

The following are constants for the conventional segment and section names:

```
#define SEG_PAGEZERO      "__PAGEZERO"  /* pagezero segment; has no
                                           protections; catches NULL
                                           references for MH_EXECUTE
                                           files */
#define SEG_TEXT          "__TEXT" /* traditional UNIX text segment */
#define SECT_TEXT         "__text" /* real text part of the text
                                      section; no headers and
                                      padding */
#define SECT_FVMLIB_INIT0 "__fvmlib_init0"  /* fvmlib initialization
                                               section */
#define SECT_FVMLIB_INIT1 "__fvmlib_init1"  /* the section following
                                               the fvmlib
                                               initialization
                                               section */
#define SEG_DATA          "__DATA" /* traditional UNIX data segment */
#define SECT_DATA         "__data" /* real initialized data section;
                                      no padding, no bss overlap */
#define SECT_BSS          "__bss"  /* real uninitialized data
                                      section; no padding */
#define SECT_COMMON       "__common" /* the section common symbols
                                        are allocated in by the link
                                        editor */
#define SEG_OBJC          "__OBJC"          /* run-time segment */
#define SECT_OBJC_SYMBOLS "__symbol_table"  /* symbol table */
#define SECT_OBJC_MODULES "__module_info"   /* (obsolete!) */
#define SECT_OBJC_STRINGS "__selector_strs" /* string table */
#define SECT_OBJC_REFS    "__selector_refs" /* string table */
#define SEG_ICON          "__ICON"          /* NeXT icon segment */
#define SECT_ICON_HEADER  "__header"        /* icon headers */
#define SECT_ICON_TIFF    "__tiff"    /* icons in TIFF format */
```

The LC_SYMTAB Load Command

The LC_SYMTAB command specifies the location and size of the symbol table information created by the compiler used for link editing and debugging. This UNIX 4.3BSD **stab**-style symbol table information is defined in the header files **nlist.h** and **stabs.h**:

```
struct symtab_command {
    unsigned long  cmd;       /* LC_SYMTAB */
    unsigned long  cmdsize;   /* sizeof(struct symtab_command) */
    unsigned long  symoff;    /* symbol table offset */
    unsigned long  nsyms;     /* number of symbol table entries */
    unsigned long  stroff;    /* string table offset */
    unsigned long  strsize;   /* string table size in bytes */
};
```

The LC_SYMTAB command contains the offsets for both the symbol table entries and the string table used by those entries. This format is different from the UNIX 4.3BSD **a.out** format: The string table offset and size are explicitly defined, and the symbol table and string tables are located at the end of the file (not after the LC_SYMTAB command).

The format of a symbol table entry is defined in the header file **nlist.h**:

```
struct nlist {
    union {
        char      *n_name;    /* for use when in-core */
        long       n_strx;    /* index into file string table */
    } n_un;
    unsigned char  n_type;    /* type flag; see below */
    unsigned char  n_sect;    /* section number or NO_SECT */
    short          n_desc;    /* see the header file stab.h */
    unsigned       n_value;   /* value of this symbol table entry
                                 (or stab offset) */
};
```

Symbols with an index into the string table of zero (**n_un.n_strx** == 0) are defined to have a null ("") name. Therefore, all string indexes to non-null names must not have a zero string length.

In the file, a symbol's **n_un.n_strx** field gives an index into the string table. An **n_strx** value of 0 indicates that no name is associated with a particular symbol table entry. The field **n_un.n_name** can be used to refer to the symbol name only if the program sets this up using **n_strx** and appropriate data from the string table.

The flag values that distinguish symbol types are defined in the header file **nlist.h**. The **n_type** field actually contains three fields, and if declared as such would be:

```
unsigned char  N_STAB:3,
               N_TYPE:4,
               N_EXT:1;
```

These fields are used by specifying the following masks:

```
#define N_STAB  0xe0  /* if any bits are set, this is a symbolic
                         debugging entry */
#define N_TYPE  0x1e  /* mask for the type bits */
#define N_EXT   0x01  /* external symbol bit; set for external
                         symbols */
```

Some of the N_STAB bits will be set if and only if the entry is a symbolic debugging entry (an **stab**)—in this case, the values for the N_TYPE bits of the **n_type** field (the entire field) are as shown in the header file **stab.h**. Normal values for the N_TYPE bits of the **n_type** field are:

```
#define N_UNDF  0x0   /* undefined; n_sect == NO_SECT */
#define N_ABS   0x2   /* absolute; n_sect == NO_SECT */
#define N_SECT  0xe   /* defined in section number n_sect */
#define N_INDR  0xa   /* indirect */
```

If the type is N_SECT, the **n_sect** field contains an ordinal of the section the symbol is defined in. The sections are numbered from 1 and refer to sections in the order in which they appear in the load commands for the file they're in. Therefore the same ordinal may refer to different sections in different files. This is the most common type of symbol.

If the type is N_INDR, the symbol is defined to be the same as another symbol. In this case the **n_value** field is an index into the string table of the other symbol's name. When the other symbol is defined, they both take on the defined type and value.

The **n_value** field for all symbol table entries (including N_STABs) gets updated by the link editor based on the value of the **n_sect** field and where the section's **n_sect** references get relocated. If the value of the **n_sect** field is NO_SECT, its **n_value** field isn't relocated by the link editor.

```
#define NO_SECT     0   /* the symbol isn't in any section */
#define MAX_SECT  255   /* 1 through 255 inclusive */
```

Common symbols are represented by undefined (N_UNDF) external (N_EXT) types whose values (**n_value**) are nonzero. In this case the value of the **n_value** field is the size in bytes of the common symbol, and the value of the **n_sect** field is NO_SECT.

The LC_THREAD and LC_UNIXTHREAD Load Commands

Thread commands contain machine-specific data structures suitable for use in the thread state primitives. The machine-specific data structures follow the **struct thread_command** or **struct unixthread_command** as follows: Each flavor of machine-specific data structure is preceded by an unsigned long constant for the flavor of that data structure and an unsigned long that's the count of longs of the size of the state data structure, and then the state data structure follows that. This triple may be repeated for many flavors.

The constants for the **flavor**, **count**, and **state** data structure definitions are expected to be in the header file **machine/thread_status.h**; these machine-specific data structure sizes must be multiples of **sizeof(long)**. The **cmdsize** reflects the total size of the **thread_command** structure and all of the sizes of the constants for the **flavor**, **count**, and **state** data structures.

```
struct thread_command {
    unsigned long  cmd;           /* LC_THREAD or LC_UNIXTHREAD */
    unsigned long  cmdsize;       /* sizeof(struct thread_command) */
    /* unsigned long  flavor         flavor of thread state */
    /* unsigned long  count          count of longs in thread state */
    /* struct XXX_thread_state state   flavor's thread state */
    /* . . . */
};
```

The LC_UNIXTHREAD command specifies an initial thread execution state for a UNIX process. For an executable object that's a UNIX process, there's one **unixthread_command** created by the link editor. A stack is created based on the UNIX rlimit for the stack. This stack will contain the command arguments and environment variables when the program is executed. The entry point is placed in the program counter in the thread state. The stack address is placed in the stack pointer by the kernel when this program is executed. The stack is created as a zero-fill on demand region when the object is launched. Then the command line and environment arguments are placed on the stack and the stack pointer in the thread state is modified.

The LC_LOADFVMLIB and LC_IDFVMLIB Commands

A fixed virtual shared library has the file type MH_FVMLIB in the Mach header, and contains the **fvmlib_command** LC_IDFVMLIB to identify the library. An object that uses a fixed virtual shared library contains the **fvmlib_command** LC_LOADFVMLIB for each library it uses:

```
struct fvmlib_command {
    unsigned long  cmd;       /* LC_IDFVMLIB or LC_LOADFVMLIB */
    unsigned long  cmdsize;   /* includes pathname string */
    struct fvmlib  fvmlib;    /* the library identification */
};
```

Fixed virtual memory shared libraries are identified by the target pathname (the name of the library as found for execution) and the minor version number:

```
struct fvmlib {
    union lc_str   name;           /* library's target pathname */
    unsigned long  minor_version;  /* library's minor version
                                      number */
};
```

The LC_LOADFVMFILE Command

The LC_LOADFVMFILE command contains a reference to a file to be loaded at the specified virtual address:

```
struct fvmfile_command {
    unsigned long  cmd;            /* LC_FVMFILE */
    unsigned long  cmdsize         /* includes pathname string */
    union lc_str   name;           /* files pathname */
    unsigned long  header_addr;    /* files virtual address */
};
```

Relocation Information

The value of a byte in a section that isn't a portion of a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a byte in a section involves a reference to an undefined external symbol, as indicated by the relocation information, the value stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the bytes in the file.

If relocation information is present, it amounts to eight bytes for each relocatable entry. The structure of a relocation entry as given in the header file **reloc.h** is as follows:

```
struct relocation_info {
    int       r_address;        /* offset in the section to what is
                                   being relocated */
    unsigned  r_symbolnum:24,   /* symbol index if r_extern == 1 or
                                   section ordinal if r_extern == 0 */
              r_pcrel:1,        /* was relocated pc-relative already */
              r_length:2,       /* 0=byte, 1=word, 2=long */
              r_extern:1,       /* doesn't include value of symbol
                                   referenced */
              r_reserved:4;     /* reserved */
};
#define R_ABS  0   /* absolute relocation type for Mach-O files */
```

The **r_address** is an offset rather than an address. For Mach-O object files this offset is from the start of the section the relocation entry is for.

If **r_extern** is 0, **r_symbolnum** is an ordinal representing the section that contains the symbol being relocated. These ordinals refer to the sections in the object file in the order in which their section structures appear in the headers of the object file they're in. The first section has the ordinal 1, the second has the ordinal 2, and so on. Therefore the same ordinal in two different object files could refer to two different sections. Furthermore, the ordinals could change when combined by the link editor. The value R_ABS is used for relocation entries of absolute symbols that need no further relocation.

To make scattered loading by the link editor work correctly, "local" relocation entries can't be used when the item to be relocated is the value of a symbol plus an offset (where the resulting expression is outside the block the link editor is moving, blocks are divided at symbol addresses). If the item is a symbol value plus offset, the link editor needs to know more than just the section in which the symbol was defined. What is needed is the actual value of the symbol without the offset, so the link editor can do the relocation correctly based on where the value of the symbol got relocated to, not the value of the expression (with the offset added to the symbol value). For Release 2.0, no "local" relocation entries are ever used when there is a nonzero offset added to a symbol. The "external" and "local" relocation entries remain unchanged.

It's assumed that a section will never be bigger than 2**24 – 1 (0x00ffffff or 16,777,215) bytes. This assumption allows the **r_address** (which is really an offset) to fit into 24 bits, and for the high bit of the **r_address** field in the **relocation_info** structure to indicate that it's really a **scattered_relocation_info** structure. Since these are only used in places where "local" relocation entries are used and not where "external" relocation entries are used, the **r_extern** field has been removed.

```
#define R_SCATTERED 0x80000000 /* mask to be applied to r_address
                                  field of a relocation_info struct
                                  to tell that it is really a
                                  scattered_relocation_info struct */
struct scattered_relocation_info {
  unsigned int r_scattered:1,  /* 1=scattered, 0=non-scattered */
               r_pcrel:1,      /* was relocated pc relative already */
               r_length:2,     /* 0=byte, 1=word, 2=long */
               r_reserved:4,   /* reserved */
               r_address:24;   /* offset in the section to what is
                                  being relocated */
  long         r_value;        /* the value the item to be relocated
                                  refers to (with no offset added) */
};
```

The Makeup of Executable Object Files

A typical executable (that is, with the filetype MH_EXECUTE) Mach-O object file produced by the link editor would contain the following components, in the order shown here:

- A Mach header
- An LC_SEGMENT load command for the **__PAGEZERO** segment
- An LC_SEGMENT load command for the **__TEXT** segment, followed by section headers for the sections in that segment. These section headers could include **__text, __fvmlib_init0**, **__fvmlib_init1**, **__const**, **__string**, **__literal8**, and **__literal4.**

- An LC_SEGMENT load command for the **__DATA** segment, followed by the section headers for the sections in that segment. These section headers could include **__data**, **__bss**, and **__common.**
- An LC_SEGMENT load command for the **__OBJC** segment, followed by the section headers for the sections in that segment. These section headers could include **__class**, **__meta_class**, **__cat_inst_meth**, **__els_meth**, **__inst_meth**, **__message_refs**, **__symbols**, **__category**, **__class_vars**, **__module_info**, and **__selector_strs.**
- An LC_SEGMENT load command for the **__LINKEDIT** segment
- An LC_SYMTAB load command
- An LC_UNIXTHREAD load command
- An LC_LOADFMVLIB load command for each shared library it uses
- The **__TEXT** segment rounded out to the segment alignment
- The **__DATA** segment rounded out to the segment alignment
- The **__OBJC** segment rounded out to the segment alignment
- All the relocation entries, if saved (normally not saved)
- All the **stab** symbol and string tables, if not stripped

You can use the **otool** command to print the contents of object files and libraries that are in Mach-O format or in UNIX 4.3BSD **a.out** format. Various options allow you to specify certain portions of the Mach-O file. For example:

-h	Print the Mach header
-l	Print the load commands
-t	Print the contents of the **__text** section (used with the **-v** flag, this disassembles the text; with the **-V** flag it also symbolically disassembles the operands)
-d	Print the contents of the **__data** section
-r	Print the relocation entries

Complete documentation for the **otool** command is contained in a UNIX manual page, which you can access through the Digital Librarian.

Additional information related to the Mach-O file format is contained in section 1 (commands), section 3 (subroutines), and section 5 (file formats and conventions) of the UNIX manual pages. You can use the following list and the Digital Librarian to find the documentation you need:

atom(1)	Converts an object file from **a.out** to Mach-O format
gdb(1)	Debugs using the GNU debugger
ld(1)	Links using the link editor
nm(1)	Prints a symbol table
otool(1)	Prints parts of an object file or library
size(1)	Prints the size of an object file
strip(1)	Removes symbols and relocation bits
getmachheaders(3)	Gets the Mach headers for an executable
getsectbyname(3)	Gets the section information for a section
getsegbyname(3)	Gets the segment command for a segment
nlist(3)	Gets entries from a name list
Mach-O(5)	Describes Mach-O assembler and link editor output
stab(5)	Describes symbol table types

15 *Building a Simple Application*

15 Building a Simple Application

Even the simplest application that presents the user with a graphic interface represents a staggering amount of programming effort. If each developer had to begin at the beginning, few applications would make it to completion. Fortunately, NeXTSTEP—through its integrated software kits and programming tools—dramatically reduces your workload in creating such applications.

NeXTSTEP's object-oriented software kits let your application benefit from numerous years of software development and testing, providing you with the elements (windows, buttons, text- and image-handling objects, and so on) that most applications require. Interface Builder dramatically simplifies the task of assembling and interconnecting these elements and helps you create new, reusable elements of your own. Overseeing the entire development process is Project Builder. Project Builder keeps track of the elements that make up your application, gives you access to other development tools, builds your application, and helps you with many other details.

The project presented in this chapter will give you a taste of application development using NeXTSTEP. You'll create a complete, though content-free, application using Interface Builder to assemble "off-the-shelf" objects from the Application Kit and then build the application using Project Builder. The objectives of this project are twofold:

- To introduce NeXTSTEP's main development tools: Interface Builder and Project Builder.
- To give you an understanding of your part in the application-development process by showing you which parts can be done entirely with Project Builder and Interface Builder.

Creating a Project

The first step in building any NeXTSTEP application is to use Project Builder to create a project. A project is a directory of files under the control of Project Builder.

Start Project Builder from the Workspace Manager, either from its location in **/NextDeveloper/Apps/ProjectBuilder** or from the dock, if its icon is there. When the application starts, it displays its main menu.

Choose New from the Project menu. The Open panel that appears has two points of interest. First is the Name field, which suggests "PB.project". This is the standard project file that Project Builder uses to record the elements and dependencies within a project. Second is the pop-up list that indicates that the type of project being built is an application.

For clarity, a new project should be created in its own directory, so let's create a directory for this project. Enter "Simple" in the Name field—overwriting "PB.project"—and press Return. (You don't have to specify **PB.project**. By default Project Builder adds this file to a new project directory.)

A Project window titled "Simple" appears.

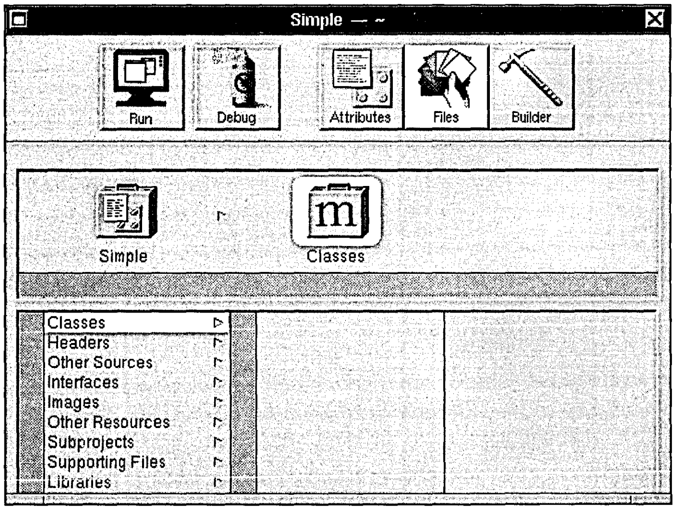


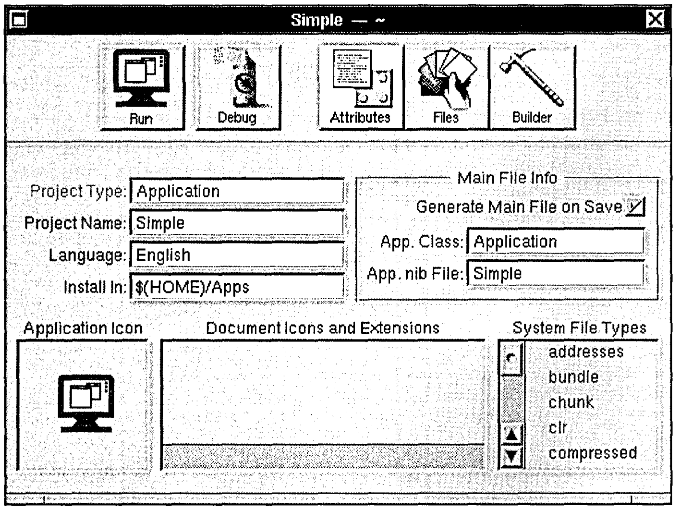
Figure 15-1. The Project Window

The five buttons at the top of the window give you access to Project Builder's main commands and displays. The buttons have these functions:

Run	Builds the application (if necessary) and then runs it.
Debug	Builds the application (if necessary) and then runs it in debugging mode using GDB and Edit.
Attributes	Shows the Attributes display, which lets you set the application's name, associated icon, installation directory, and other attributes.
Files	Shows the Files display, which gives you access to the files that make up the project.
Builder	Shows the Builder display. This display lets you specify compiler and linker options and view messages generated during the build process.

Let's take a look at the three different displays.

Click the Attributes button. This display lets you set some of the global attributes of your project, such as its name, the icon the application displays in the workspace, where the finished application will be installed in the file system, and other features:



Project Type: Application
Project Name: Simple
Language: English
Install In: $(HOME)/Apps

Generate Main File on Save ✓
App. Class: Application
App. nib File: Simple

addresses
bundle
chunk
clr
compressed

Figure 15-2. Project Builder's Attributes Display

You'll learn more about the Attributes display during the course of this project and the others in the following chapters.

Click the Files button. The Files display gives you an organized view of the files that make up your application. (See Figure 15-1 above for an illustration.) The left column lists the types of source files, and, for a selected type, the right column (or columns) displays the names of any files of that type that your project contains.

Finally, click the Builder button.

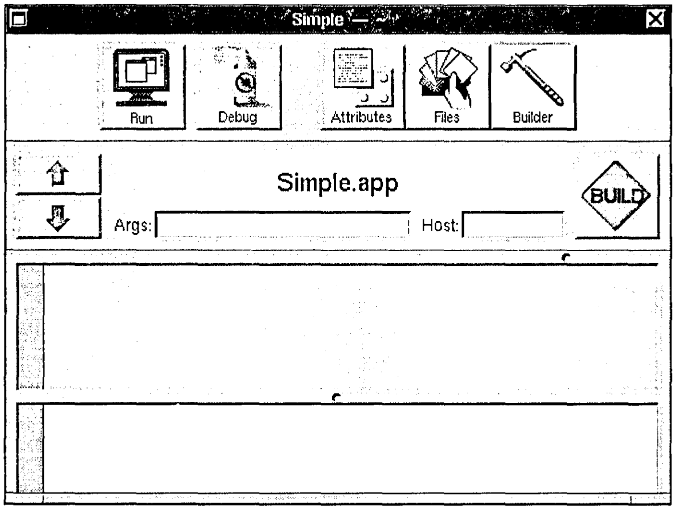


Figure 15-3. Project Builder's Builder Display

You use this display to control how your application is built. For example, using the Args field, you can specify whether a debugging or an optimized version of the application will be built. Using the Host field, you can specify that the application be compiled and linked on some other computer on the network, thus reducing the load on your computer.

Return to the Files display by clicking the Files button. If you click the different entries in the Files display, you'll notice that Project Builder has already created these files for the Simple project:

Type	**File**
Other Sources	Simple_main.m
Interfaces	Simple.nib
Supporting Files	Makefile

In addition, the standard shared libraries, Media_s and NeXT_s, are listed under Libraries. Their entries are listed in gray since you can't remove them.

Simple_main.m is the project's main program file, the source file that contains the entry point (that is, the **main()** function) for the Simple application. You can open this (or any) file listed in the Files display by double-clicking the file name in the browser. Double-click **Simple_main.m** in the browser to see how this works. Leave the file unchanged, however, since Project Builder maintains it for you. When you're through viewing the file, close the Edit window.

Simple.nib is a template interface file that Project Builder has added to the project. In "Creating the User Interface" below, we'll examine and edit this file, so don't open it yet. The flag image next to the file's name indicates that its contents may require "localization," that is, adaptation of its text, images, and sounds for speakers of different languages. Project Builder helps you create and maintain localized versions of your application.

Finally, **Makefile**, a specification file for the UNIX **make** utility, lists the files and dependencies for building your project. As with **Simple_main.m**, don't modify the contents of this file; Project Builder maintains it for you.

With this short introduction to Project Builder's main features, you are ready to move on to the next step: assembling your application's user interface.

Creating the User Interface

As a starting point for an application's user interface, Project Builder supplies new projects with a template user interface file. In our case, this file is listed as **Simple.nib** under the Interfaces entry of Project Builder's Files display. You use Interface Builder to modify the contents of this file.

Switch to Project Builder's Files display and double-click **Simple.nib** to start Interface Builder. Interface Builder starts and displays several windows. Before beginning work on the interface, let's take a short look at Interface Builder itself.

As an application that helps you build applications, Interface Builder displays both its own windows and those of the application under construction. In this case, the window titled "My Window" and the menu titled "Simple" belong to your application; the other windows are Interface Builder's. At the upper left of the screen is Interface Builder's main menu, at the upper right is the Palettes window, and at the lower left is a window titled "Simple.nib". This last window is referred to as the File window, since it has the title of the nib file and it gives you access to the contents of that file.

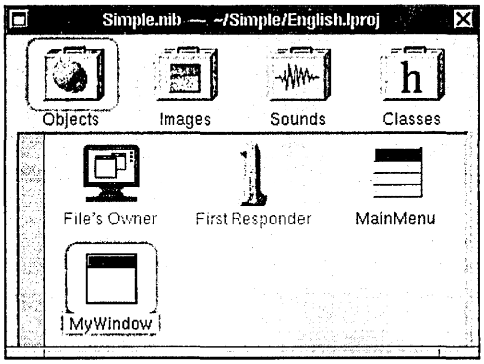


Figure 15-4. The File Window

A File window's title displays the name of the nib file and its directory location. Below the title is a row of icons. These icons correspond to the objects and resources in the nib file. The four icons give you access to these displays:

Display	**Use**
Objects	Shows the top-level objects within your application
Images	Displays the image resources available to your application.
Sounds	Displays the sound resources available to your application.
Classes	Displays a hierarchical listing of the classes available to your application.

Initially, the first icon is highlighted, indicating that the File window shows the Objects display. In the upper left corner of the display is an icon representing the nib file's owner. The two objects named "MainMenu" and "MyWindow" correspond to the Menu and Window objects in the nib file. The object titled "First Responder" represents the object that at run time has first responder status within MyWindow.

Within the File window, you can edit an object's name by selecting the object and then clicking its 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. For now, however, leave the default names.

Adding and Editing 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 features of the Palettes window.

The Palettes window by default has four distinct displays, represented by the four buttons near the top. (Note, however, that more palettes can be loaded into the Palettes window—see Chapter 18 for more information.) The left button gives you access to Menus and MenuCells, the next to Windows, the next to Basic Views, and the last to Scrolling Views:

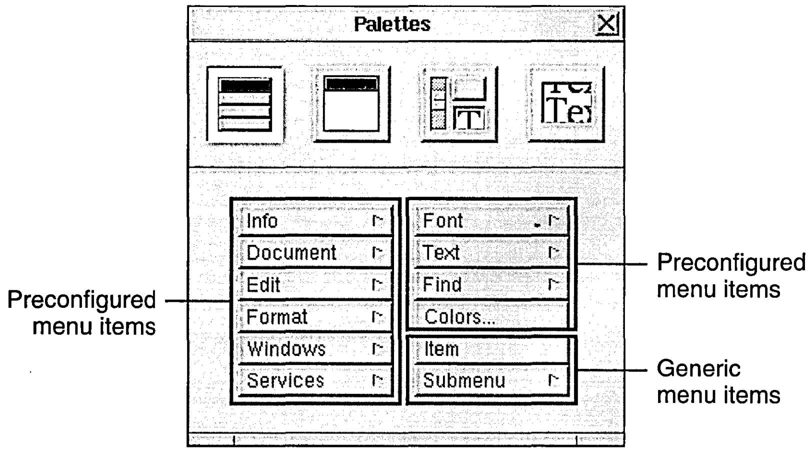


Figure 15-5. Menu Palette

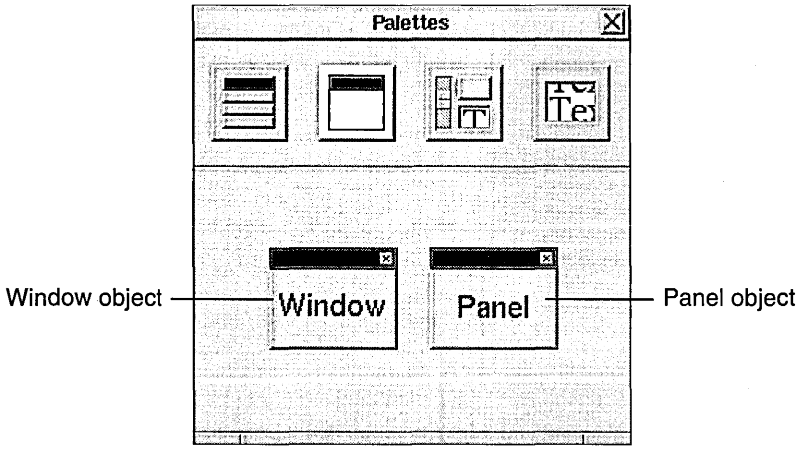


Figure 15-6. Window Palette

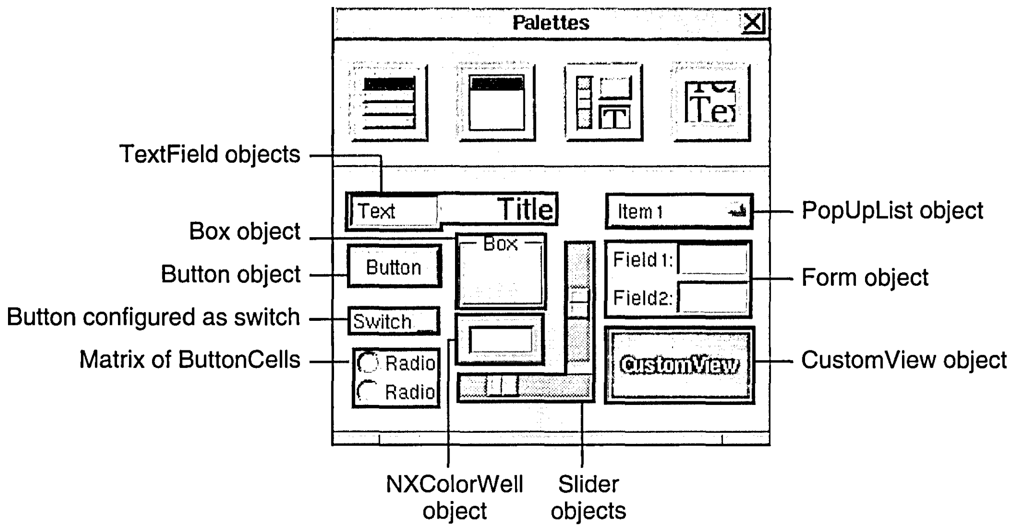


Figure 15-7. Basic Views Palette

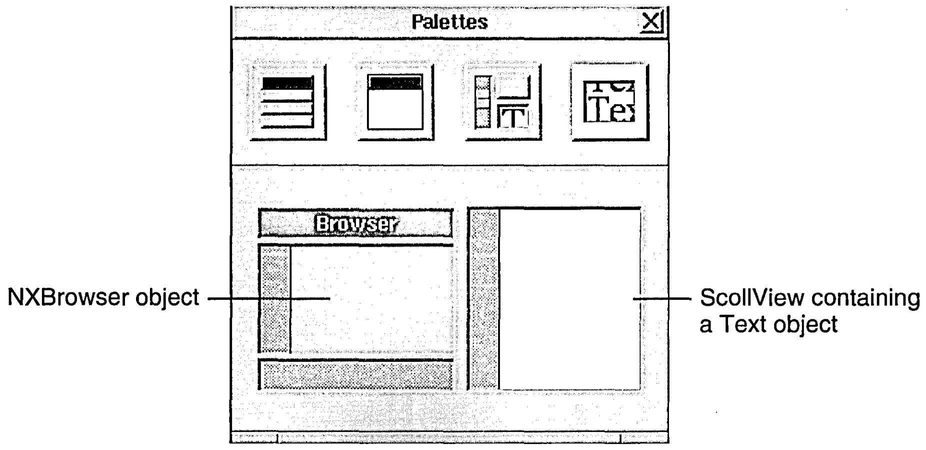


Figure 15-8. Scrolling Views Palette

Let's add some objects to the application. First, make sure the Basic Views palette is displayed. Drag a Button object from the Palettes window into your application's standard window.

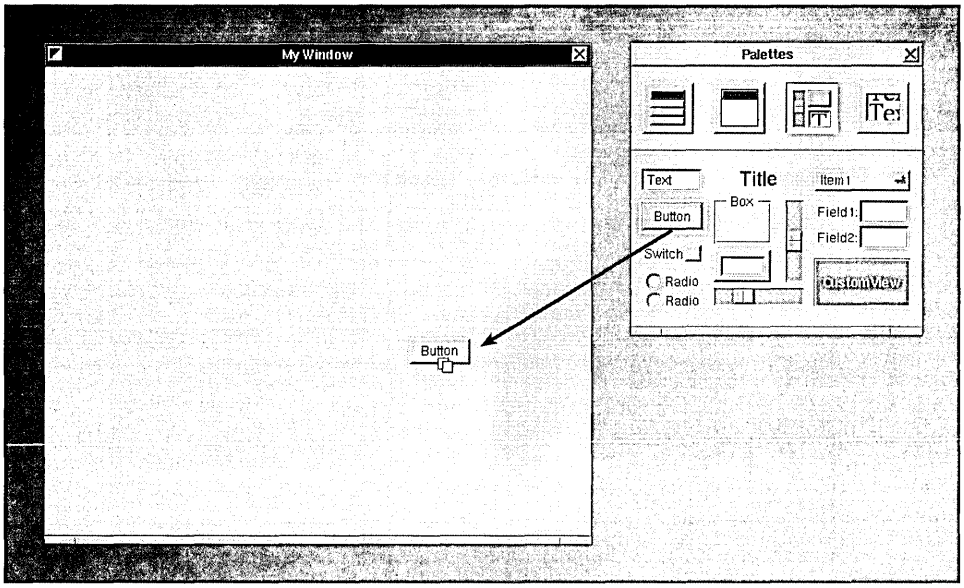


Figure 15-9. Adding a Button to Your Application

Notice that as you drag the button over the destination window, the cursor changes to the copy cursor, indicating that if you release the mouse button, the object will be copied into the window. Release the mouse button, and the Button object drops into the window. Eight small gray squares, or *control points*, appear around the button. These control points indicate that the button is selected. 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.

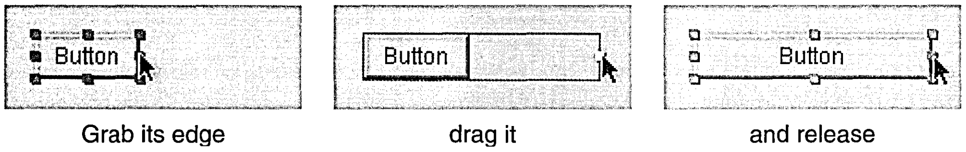


Figure 15-10. Resizing an Object

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 only vertically or horizontally by Command-dragging it. If you start Command-dragging vertically, for example, no horizontal motion is possible until you release the mouse button and begin dragging again.

Let's add another button to the window. You could drag a second button from the Palettes window, but instead, try copying and pasting the existing button. First make sure the button is selected (its control points should be visible) and then choose the Copy and then Paste commands from the Edit menu. A second button appears overlapping the first. Notice that the second button is now selected and the first is not. Drag the second button to one side of the first.

When you select one 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.

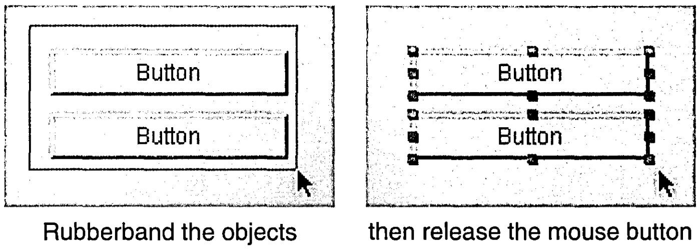


Figure 15-11. Selecting Objects by Rubberbanding

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. Edit the title of one button to read "On".

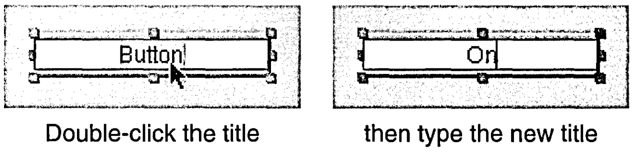
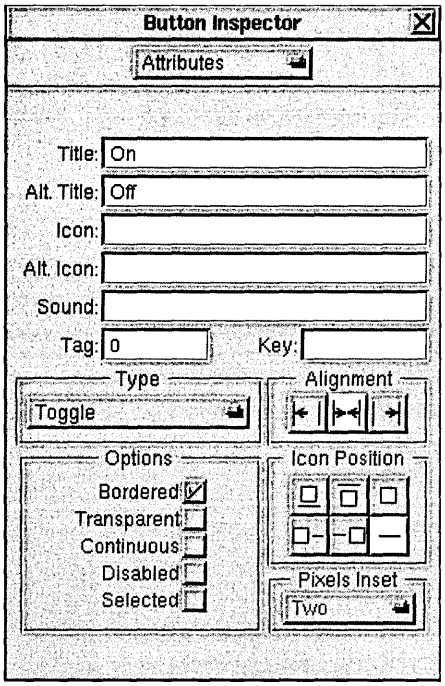


Figure 15-12. Editing an Object's Text

To edit an object's attributes that can't be easily manipulated graphically, Interface Builder provides an Inspector panel for the particular object. The Button Inspector, for example, lets you set the type and appearance of a button, among other things.

To display the Button Inspector, first make sure the On button is selected and then choose the Inspector command from the Tools menu. The Inspector panel appears. This panel's title changes to reflect the object that is being inspected; it reads "Button Inspector" since the button is selected. The Inspector panel has multiple displays accessed by the pop-up list at the top of the panel. The other displays will be discussed in this and later chapters; for now, let's work with the Attributes display.

Using the Button Inspector, let's configure the On button to be a button that toggles between two states labeled "On" and "Off".

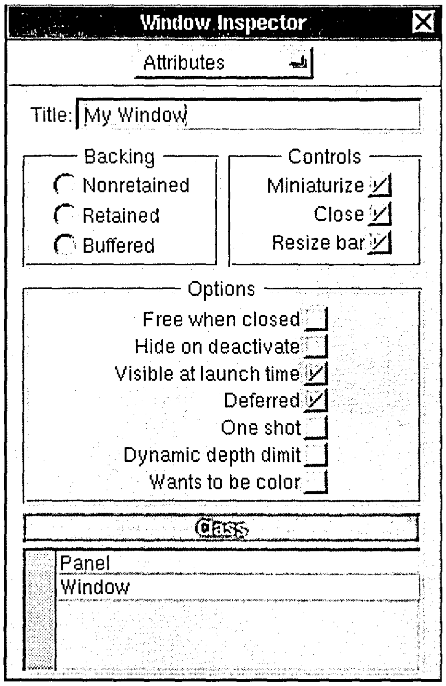


Title: On
Alt. Title: Off
Icon:
Alt. Icon:
Sound:
Tag: 0 Key:
Type: Toggle
Options
Bordered ✓
Transparent ☐
Continuous ☐
Disabled ☐
Selected ☐
Pixels Inset: Two

Figure 15-13. The Button Inspector

In the Inspector panel, type "Off" in the text field labeled "Alt. Title" (Alternate Title). Next, set the button type by pressing the Type pop-up list and dragging to the Toggle option. You can check the operation of this button in a moment when you test the interface.

Before leaving the Inspector panel, let's use it to change the title of the application's standard window. Select the window titled "MyWindow" (by clicking anywhere within its boundaries or by double-clicking its icon in the File window). The display in the Inspector panel changes from the Button Inspector to the Window Inspector. Notice that the Window Inspector (as shown in Figure 15-13) lets you set the window's title, class, and other attributes.


Window Inspector

Attributes

Title: My Window

Backing
- Nonretained
- Retained
- Buffered

Controls
- Miniaturize ✓
- Close ✓
- Resize bar ✓

Options
- Free when closed
- Hide on deactivate
- Visible at launch time ✓
- Deferred ✓
- One shot
- Dynamic depth dimit
- Wants to be color

Class
- Panel
- Window

Figure 15-14. The Window Inspector

Change the window's title to "Test Window" or another title of your choice. Notice that when you begin to alter the window's title, the Inspector panel's close button changes to display a partially drawn "X," indicating that your changes haven't yet been applied to your application's window. When you press Return, Interface Builder applies the changes to the window's title.

Before continuing, choose the Save command from the Document menu to save the work you've done so far to the nib file, **Simple.nib**.

Laying Out the Interface

You can arrange the top-level components of your application—its windows and panels—through direct manipulation. For example, to specify where a window will appear at run time, simply drag it to that position within Interface Builder. (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 runs. If it will be, resize it with the resize bar as you normally would. If at run time its size is fixed (as with an application's Info panel), you have to make it temporarily resizable within Interface Builder by clicking the resize button ◩ in the title bar.

Interface Builder provides a large selection of layout tools to help you arrange objects within your application's windows. To experiment with these tools, arrange the two buttons in your application's window so that one partially covers the other, and then open the Layout menu (choose Format from the main menu, and then choose Layout). Select one button 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 it contents.

Select both buttons (you could "rubberband" them or click one button and then hold down Shift while you click the other) and choose Same Size. One button is resized to match the other button. (The object you select last is resized to match the size of the object that's selected first, unless this would cause it to be resized to less than its minimum size.)

Now, with both buttons selected, choose Group. This command has two effects: It visually groups selected objects by surrounding them with a box, and it makes the selected objects subviews of the surrounding box. Notice that if you move the surrounding box, the buttons, being subviews, move with it whether or not they're selected. To move a button within the box, double-click within the box. A gray border appears indicating that the editing focus is now within the box. Once the focus is on the box's contents, you can manipulate the individual buttons as you normally would. This is the pattern for editing grouped objects. For example, if a button is grouped in a box that in turn is grouped within another box, you can edit the button's title by double-clicking the outer box, then the inner box, and then the button itself. To remove a surrounding box without destroying its contents, select the box and then choose the Ungroup command.

You can also group objects within a ScrollView. Select both buttons again and choose the Group in ScrollView command from the Layout menu. A ScrollView appears around the two buttons; however, no scroll knobs are visible. Again, double-clicking within the ScrollView allows you to manipulate the grouped objects individually.

Double-clicking within the ScrollView changes the editing focus to the ScrollView's document view within the ScrollView. Once the focus is on the document view, you can manipulate the grouped objects individually and you can resize the document view. Notice that as you move the cursor toward the top or right side of the ScrollView, the cursor's image changes to that of the resizing cursor. When this cursor image is displayed, you can press the mouse button and drag the side of the document view to change the view's size. Experiment with this feature and notice how resizing the document view affects the sized of the scroll knobs.

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. Add three or four switches from the Palettes window to your application's window and experiment with these commands.

The Turn Grid On command turns on an alignment feature in all your application's windows, making it easier to create pleasing layouts. Choose the Turn Grid On command and then drag one of the switches. Notice that the switch 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 visible only 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.

The Alignment command opens a panel that affects how the alignment commands in the Layout menu work.

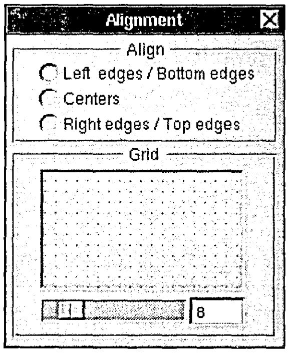


Figure 15-15. The Alignment Panel

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 slider in the Grid group, you can set the spacing of the alignment grid. Experiment with these controls, if you wish.

Any of the Controls in the Views palette—in other words, the Slider, TextField, 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 Basic Views palette, 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 other 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 run the application in test mode, choose the Test Interface command from the Document 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 are in the way the Hide and Quit menu commands operate. 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.

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 Document menu to save your work.

Preparing to Compile the Application

Before compiling the application, let's take a look at the pieces Project Builder has provided. If you look in the project directory, you'll see these entries:

- *language*.lproj(A directory where *language* is English, French, or another language.)
- Makefile
- PB.gdbinit
- PB.project
- Simple.iconheader
- Simple_main.m

The ".lproj" directory contains files that are specific to a particular language or cultural context. In the case of the Simple application, only its nib file, **Simple.nib**, has elements (menu commands and button titles, for instance) that would have to change if the application were to be in a different language environment. Thus, this directory contains only the nib file. (It may also contain a backup file. A backup file is marked with a trailing tilde character (~) and contains the previous version of the file. For example, the backup file for **Simple.nib** is **Simple.nib~**.)

Makefile, the file that coordinates the compilation process, is constructed from information in **PB.project**. Don't make changes to this file; Project Builder maintains this file for you. (However, by adding a **Makefile.preamble** or **Makefile.postamble** file, you can supplement the instructions in the standard makefile.)

PB.gdbinit contains initialization commands for the debugger, GDB. Again, don't alter this file since Project Builder maintains it for you.

PB.project contains a simple ASCII listing of your project's attributes, such as its name, installation directory, and source files. Project Builder uses this information to construct the makefile, among other things.

Simple.iconheader contains information that the Workspace Manager will use to relate icons with the application and its documents.

The last file, **Simple_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. Project Builder generates the makefile and fills in the names of your application's source files in the appropriate spots:

```
#
# Generated by the NeXT Project Builder.
#
# NOTE: Do NOT change this file -- Project Builder maintains it.
#
# Put all of your customizations in files called Makefile.preamble
# and Makefile.postamble (both optional), and Makefile will
# include them.
#

NAME = Simple

PROJECTVERSION = 1.1
LANGUAGE = English

LOCAL_RESOURCES = Simple.nib

MFILES = Simple_main.m

OTHERSRCS = Makefile

MAKEFILEDIR = /NextDeveloper/Makefiles/app
INSTALLDIR = $(HOME)/Apps
INSTALLFLAGS = -c -s -m 755
SOURCEMODE = 444

ICONSECTIONS = -sectcreate __ICON app
        /usr/lib/NextStep/Workspace.app/application.tiff

LIBS = -lMedia_s -lNeXT_s
DEBUG_LIBS = $(LIBS)
PROF_LIBS = $(LIBS)

-include Makefile.preamble
include $(MAKEFILEDIR)/app.make
-include Makefile.postamble
-include Makefile.dependencies
```

You shouldn't alter this makefile; Project Builder maintains it for you. Notice that it lists the name of your application and the source files that are specific to it. It lists the libraries that the linker uses 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
- app.make
- Makefile.postamble
- Makefile.dependencies

app.make is always included; the other files are included only if they're present. No error occurs if they're not. **app.make** is the standard NeXT makefile. The ability to include other files lets you add additional rules to this standard makefile.

Simple_main.m

This file contains your application's **main()** function:

```
/* Generated by the NeXT Project Builder
   NOTE: Do NOT change this file -- Project Builder maintains it.
*/

#import <appkit/Application.h>

void main(int argc, char *argv[]) {
    [Application new];
    if ([NXApp loadNibSection:"Simple.nib" owner:NXApp withNames:NO])
            [NXApp run];
    [NXApp free];
    exit(0);
}
```

This file starts by importing **appkit/Application.h** for its declaration of the Application class and the NXApp global variable, which refers to the Application object.

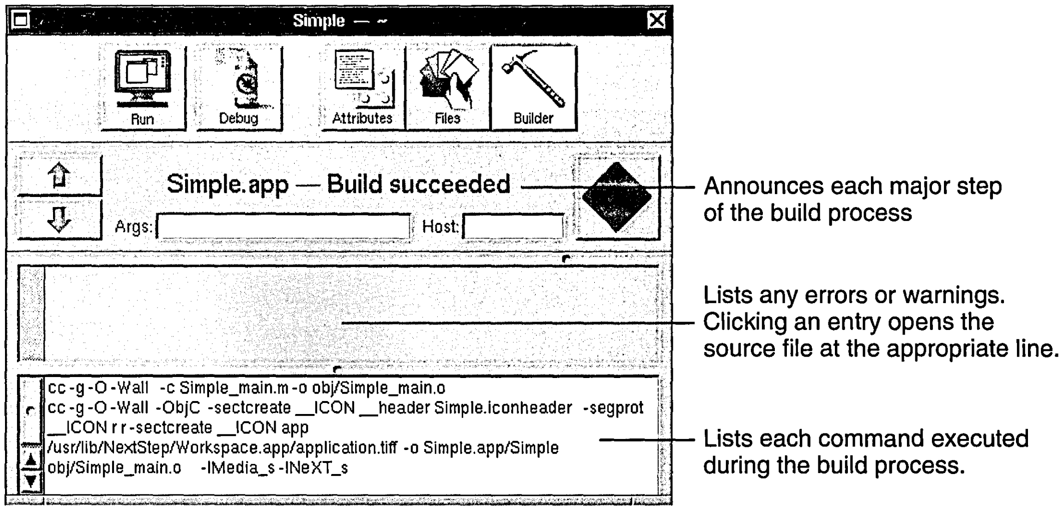
Note: A fast way to determine where a specific constant, function, or method is declared is to use Digital Librarian to search the files in **/NextDeveloper/Headers**.

It then creates a new Application object and sends a **loadNibSection:owner:withNames:** message to it to load the nib file. The **loadNibSection:owner:withNames:** method locates the correct nib file based on the user's current language preference. (See the Application class description for more information.) Assuming the nib file is found, the objects archived in it are loaded, and then the Application object is sent a run message. At this point the application becomes responsive to the user. When the user chooses the Quit 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. You can compile and run the application in one step by clicking Project Builder's Run button.

When you click the Run button, Project Builder switches to the Builder display and begins building your application. While the build proceeds, its progress is reflected in various ways, as indicated in Figure 15-16.


Simple — ~
Run
Debug
Attributes
Files
Builder
Simple.app — Build succeeded
Args:
Host:
Announces each major step of the build process
Lists any errors or warnings. Clicking an entry opens the source file at the appropriate line.
Lists each command executed during the build process.

```
cc -g -O -Wall  -c Simple_main.m -o obj/Simple_main.o
cc -g -O -Wall  -ObjC  -sectcreate __ICON __header Simple.iconheader  -segprot
__ICON r r -sectcreate __ICON app
/usr/lib/NextStep/Workspace.app/application.tiff -o Simple.app/Simple
obj/Simple_main.o   -lMedia_s -lNeXT_s
```

Figure 15-16. Building Simple.app

Running the Application

When the building process is finished, your application begins running. When your application's windows appear, you can verify its operation.

Although limited in scope, this simple 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 rebuilding the application. Rebuilding will take little time, since changing the interface alters only the nib file, not any files that must be recompiled.

16 *Building a One-Button Calculator*

16 *Building a One-Button Calculator*

This chapter describes how to build a simple calculator using many of the techniques introduced in the previous chapter. 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, but its first task will be to convert Celsius temperatures to Fahrenheit. As a temperature converter, the application's calculator window looks like this:

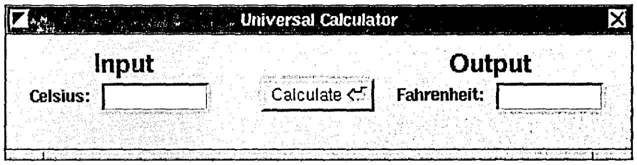


Figure 16-1. The Universal Calculator

The user enters a Celsius temperature in the left text field, then presses Return or clicks the Calculate 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 TextField object, performs the calculation, and then sends a message to the right TextField object to set its value to the result.

Creating the Interface

As you did in the first project, create a new project by choosing New from Project Builder's Project menu. Name this new project "Calculator" and save it in your home directory.

When the project window appears, double-click **Calculator.nib** to open the interface file. (**Calculator.nib** is listed under Interfaces in Project Builder's Files display.)

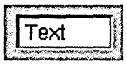
Interface Builder becomes active and opens the template file for this new project. If the interface file for the previous project is still open you'll notice that Calculator's File window opens and overlaps the File window for Simple. By allowing multiple nib files to be open at the same time, Interface 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 nib file associated with the File window the current nib file. In Interface Builder, commands such as Save or Close operate on the current nib file.

Since Simple's nib file is finished, close it by making it current and then choosing the Close command from the File menu.

Next, drag the interface objects shown in Figure 16-1 above from the Basic Views palette to the application's standard window. (You'll find it easier to align the different objects if you first turn on the grid.) You'll be using only two types of objects—Buttons and TextFields—although the TextFields will be configured as both titles and editable text fields:

Title text field



Editable text field

Since the input and output fields are nearly identical, it's fastest to configure one first, then duplicate it and modify the copy to create the other field. Drag an editable text field into the window and stretch it a bit horizontally. Delete the word "Text" by double-clicking it and pressing the Delete key. Now drag in two title text fields and place one above and the other to the left of the editable text field. Edit the titles to match those in Figure 16-1 by double-clicking them in turn. You can also change their fonts and sizes using the Font panel, which is accessible through Interface Builder's Format menu.

Resize the window so that it resembles the window in Figure 16-1 above. Now open the Window Inspector by dragging to Attributes in the Inspector panel's pop-up list. Change the window's title to "Universal Calculator."

With the exception of the Return icon in the Calculate button, the Universal Calculator window in your application should look identical to the one in the figure above. To add the icon, open the Images display (shown in Figure 16-2) by clicking the Images button in the File window.

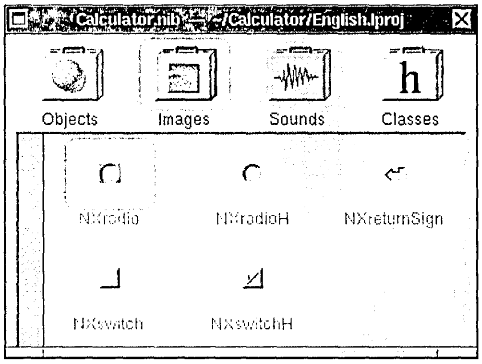


Figure 16-2. The Images Display of the File Window

This display shows the images that are used throughout the Application Kit and lets you add new images by dragging them in from the File Viewer. Once an image is displayed in this window, you can drag it onto Button objects in your application.

Drag an NXreturnSign image from the File window to the Calculate button in the Universal Calculator. When the cursor intersects some part of the button, it changes to the link cursor, indicating that releasing the mouse button will assign the image 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 buttons that are grouped in the Icon Position cluster. Try several different placements if you like. If you place the icon above or below the title, the Calculate 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 or use the Size to Fit command.

Defining the Calculator Class

The Calculator object is this application's control center. The Calculate 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 TextField objects to ascertain the input value and set the output value. We'll use the Classes display of the File window to design the Calculator class to handle these tasks. Click the Classes icon at the top of the File window.

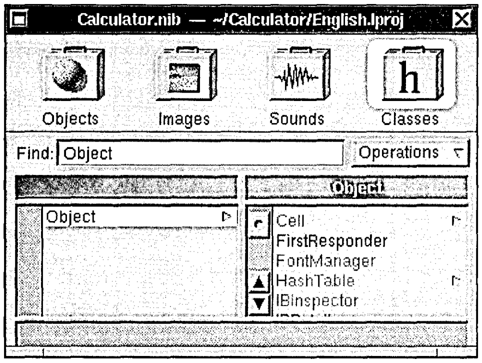


Figure 16-3. The Classes Display

This display shows a hierarchy of the classes available to your application. With the exception of the First Responder entry, class names are displayed in gray, indicating that these classes can't be edited. (First Responder, as mentioned previously, is not a particular class, but the class of an object that has first-responder status in a window.)

Notice that the Inspector panel 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.

Returning to the File window, select the Application class entry. (A quick way to locate a class is to type its name in the Find field and press Return.) Its superclass, Responder, is displayed as the title of one browser column.

With the Application class selected in the Classes window, the Class Inspector displays an Application object's outlet (**delegate**) and action methods (such as **hide:** and **terminate:**). Again, these entries are displayed in gray since they aren't editable.

Using the File 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 File 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 "MyObject" appears in the right column. The class name also appears in the text field in the Class inspector. Edit this field to read

"Calculator" and press Return. Notice that the File 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 TextField objects a Calculator object sends messages to. Make sure the Outlets button in the Class Inspector is highlighted and then enter "inputField" in the text field below the button. Click the Add Outlet button; the new outlet appears in the Outlets list. Again, it's in black, indicating that you can rename or remove this outlet. In the same way add an outlet named "outputField".

A Calculator object also needs to respond to an action message from the Calculate button in the application. Let's specify this new action method. Click the Actions radio button so that it is highlighted and then enter "calculate:" in the text field at the bottom of the panel. Click Add Action to add this method name to those displayed in the Actions list. (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. The Class Inspector should look like this:

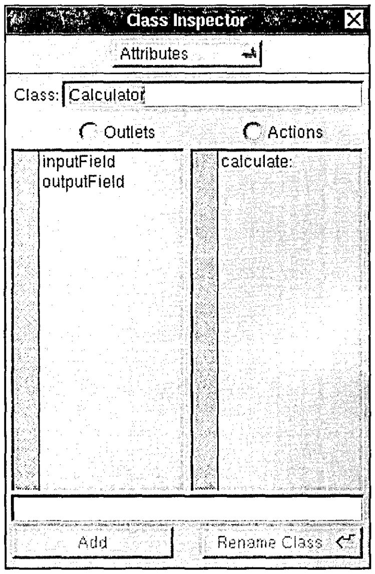


Figure 16-4. Calculator Class Interface

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.

In the File window, drag to the Unparse button in the pull-down list. A panel opens asking if you want to create **Calculator.[hm]** (a shorthand for **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]** to the project. Again, confirm that you do. Project Builder's window comes forward and shows you that the Calculator class has been added to the project.

In this project, you defined a class and had Interface Builder write template files for it. Interface Builder's File window can also be used to import the class declaration 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 the File Viewer into the File window. Interface Builder then parses the file and adds 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 also drag the class files into the Files display of Project Builder's project 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 Interface Builder's File window and then drag to the Instantiate button in the pull-down list. When you release the mouse button, the File window switches to the Objects display, and a new object appears. This icon, titled "Calculator," 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 one of the predefined connections.

When a user chooses the Hide command, an application removes all but its application icon from the screen. The MenuCell titled "Hide" sends the message and the Application object's **hide:** method performs the operation. To see this connection, click the Hide command in Calculator's main menu. Drag to Connections in the Inspector panel's pop-up list to reveal the Connections display for a MenuCell:

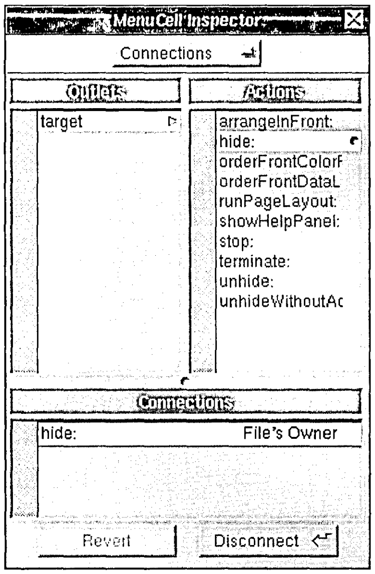


Figure 16-5. The Connections Display

The left column shows the MenuCell's sole outlet, **target**. The right column lists the action messages that the target object—in this case an Application object—recognizes. Notice that the entry **hide:** is highlighted and is marked with a small dimple. The dimple indicates that a connection using this action message has been previously established. The list titled "Connections" near the bottom of the panel summarizes the connections for the inspected object.

To see a graphic depiction of the connection, click the entry in the Connections list. The connection is displayed in the workspace by a black line drawn between the MenuCell that sends the action message and the File's Owner. Figure 16-6 shows this connection.

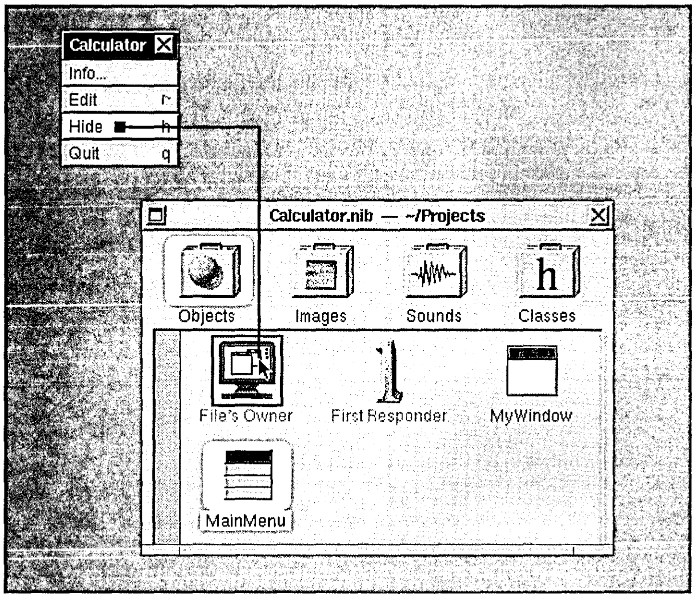


Figure 16-6. Displaying a Connection

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.

Now that you've seen how connections are indicated, let's create some in the calculator application. First, let's connect the Celsius TextField to the Calculate button so that when the user presses Return after entering a Celsius value, the button will act as if it had been clicked. Control-drag from the Celsius TextField toward the Calculate button. You'll notice that a black line trails from the cursor. When the cursor overlaps the Calculate button, a box appears around the button. Release the mouse button. The source and destination of the connection are now identified, and the TextField Inspector lists the TextField'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. The new connection is listed in the lower part of the Inspector panel.

When the user clicks the Calculate button (or it receives a **performClick:** message), the Calculator object should receive a **calculate:** message. To identify the source and destination of this connection, Control-drag a connecting line from the Calculate button to the Calculator icon in the File window. In the Button Inspector, establish that the Button's target receives a **calculate:** action message.

Next, you have to connect the Calculator object's outlets to the appropriate TextField objects. Control-drag a line from the Calculator icon in the File window toward the Celsius TextField object.

The CustomObject Inspector shows a Calculator object's two outlets, **inputField** and **outputField**. Select **inputField** and click Connect. Notice that a dimple appears next to the **inputField** listing in the Inspector panel, indicating that the connection has been made.

Following the same steps, connect the **outputField** outlet to the Fahrenheit TextField.

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 nib 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 Calculate button), but of course no calculation takes place. For that, we have to define 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, return to Project Builder and double-click **Calculator.h** and **Calculator.m**, which you'll find in the Files display under Header and Classes. The Edit application opens these files. 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**:

```
#import <appkit/appkit.h>

@interface Calculator:Object
{
    id  inputField;
    id  outputField;
}

- calculate: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 **id**s of the calculator window's input and output TextFields. Also, as listed in the class editor, a Calculator object declares the **calculate:** action method.

Calculator.m

Calculator.m will contain the implementation of the Calculator class:

```
#import "Calculator.h"

@implementation Calculator

- calculate:sender
{
    return self;
}

@end
```

The **calculate:** method must send a message to the object referred to by its **inputField** variable to retrieve the Celsius value, calculate the Fahrenheit equivalent, and then send a message to the object referred to by its **outputField** variable to set the value it displays. One implementation of this method looks like this:

```
- calculate:sender
{
    float degreesF;

    [inputField selectText:self];
    degreesF = ((9.0 * [inputField floatValue])/ 5.0) + 32.0;
    [outputField setFloatValue:degreesF];
    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. 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. Finally, save the file. You're now ready to compile and test the application.

Testing the Application

To compile and run the calculator application, click Run in Project Builder's project window.

If any errors are detected while the application is being built, they will be listed in the summary view of the Project window. Click an entry and Edit opens the file to the appropriate line, making it convenient to correct the problem. (You may want to introduce an error, just to see how this works!)

Once the application has been successfully compiled and linked, it begins to run. Test its features to verify that they all work properly.

Modifying the Calculator

So far, the Universal Calculator can handle any calculation—as long as it's converting degrees Celsius to Fahrenheit. The rest of this chapter describes how to add to the calculator's functionality and, in passing, introduces several features concerning menus and submenus. The final sections of this project demonstrate how to add icons and sounds to an application.

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 by dragging 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 16-7.

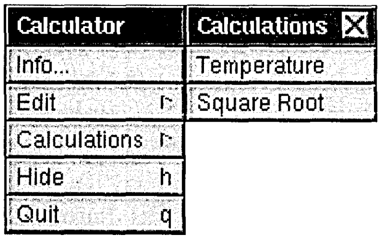
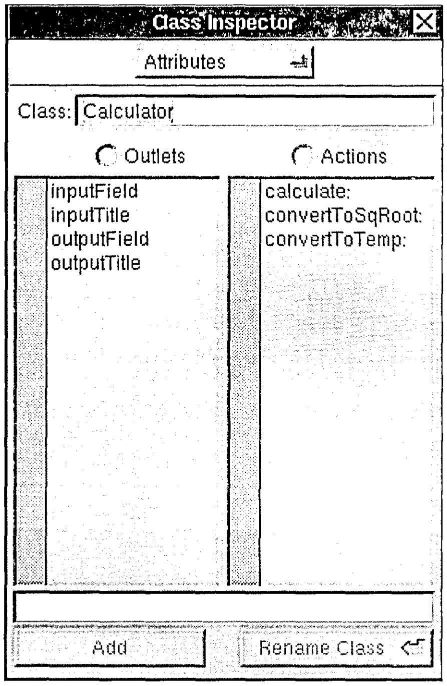


Figure 16-7. The Menu and Submenu

Now, select the Calculator class in the Classes display of the File window. To edit the class definition, open the Class Inspector (by choosing the Inspector command from Interface Builder's Tools menu). The revised Calculator object must respond to action messages from the new submenu, so let's add **convertToTemp:** and **convertToSqRoot:** methods. It will also need to send messages to the TextFields that titles the input and output fields, so let's add **inputTitle** and **outputTitle** outlets. Figure 16-8 shows how the Class Inspector should look after you make these changes to the Calculator class interface.



Class: Calculator

Outlets	Actions
inputField	calculate:
inputTitle	convertToSqRoot:
outputField	convertToTemp:
outputTitle	

Figure 16-8. 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 Calculator object in the Objects display of the File window. Double-click the **convertToTemp:** entry in the Inspector panel to establish the connection. Likewise, specify that the Square Root submenu item sends a **convertToSqRoot:** message to the Calculator object.

Now, connect the Calculator's **inputTitle** and **outputTitle** outlets to the proper TextFields in the Calculator window. (The Calculator object will send messages to these objects to change their text from "Celsius" and "Fahrenheit" to "x" and "sqrt(x)", as the user picks one or the other type of calculation.) Control-drag from the Calculator object in the File window to the TextField that reads “Celsius”. Double-click **inputTitle** in the Connections inspector to establish the connection. Follow the same process to connect the **outputTitle** outlet to the TextField currently titled “Fahrenheit”.

Finally, use the TextField inspector’s alignment buttons to specify that the text in these fields is right aligned. In this way, although a title’s text may change from “Celsius” to “x”, it will stay visually associated with the input field it labels.

The revised interface is complete; the only changes that remain affect the Calculator class files. The next two sections describe the changes you need to make.

Modifying Calculator.h

The new calculator is designed either to convert temperatures or to 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. The **inputTitle** and **outputTitle** instance variables also need to be declared. These changes add eight lines to the **Calculator.h** file. The lines you need to add are shown in bold:

```
#import <appkit/appkit.h>

#define TEMP 1
#define SQROOT 2

@interface Calculator : Object
{
    id inputField;
    id outputField;
    id inputTitle;
    id outputTitle;
    int calcType;
}

- init;
- calculate:sender;
- convertToTemp:sender;
- convertToSqRoot:sender;

@end
```

Modifying Calculator.m

The implementation file must be modified in three ways. It needs an initialization method to establish the value of the **calcType** instance variable (and thus the calculator's initial state). The **init** method below handles this initialization. When the calculator first appears, it will be configured to perform temperature conversions. It also must be modified so that the **calculate:** method performs the proper calculation according to the calculator's current state. 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.

```
#import "Calculator.h"

@implementation Calculator

- init
{
    [super init];
    calcType = TEMP;
    return self;
}

- calculate:sender
{
        [inputField selectText:self];
        if (calcType == TEMP) {
            float degreesF;
            degreesF = ((9.0 * [inputField floatValue])/5.0) + 32.0;
            [outputField setFloatValue:degreesF];
        } else if (calcType == SQROOT) {
            double sqRoot;
            sqRoot = sqrt([inputField doubleValue]);
            [outputField setDoubleValue:sqRoot];
        }
        return self;
}
```

```
- convertToTemp:sender
{
    calcType = TEMP;
    [inputTitle setStringValue:"Celsius:"];
    [outputTitle setStringValue:"Fahrenheit:"];
    [outputField setStringValue:""];
    [inputField selectText:self];
    return self;
}

- convertToSqRoot:sender
{
    calcType = SQROOT;
    [inputTitle setStringValue:"x:"];
    [outputTitle setStringValue:"sqrt(x):"];
    [outputField setStringValue:""];
    [inputField selectText:self];
    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 inspector and those in the Calculator class definition files. Use Interface Builder to check the method and variable names in the Class Inspector panel against those in **Calculator.h** and **Calculator.m**.

Adding an Icon

With the Images display of the File window, you can access existing system images, as illustrated earlier in this project, or you can create images from data in either TIFF (Tag Image File Format) or EPS (Encapsulated PostScript) file format. Once you import the image, it can be assigned to Button objects in your application. Figure 16-9 shows some examples of buttons that display icons.

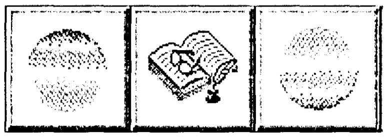


Figure 16-9. Icons and Buttons

To see how this works, click the Images suitcase in the File window to display 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 image to this window. Using the File Viewer, 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 File Viewer to Interface Builder's File window. When you release the mouse button, Interface Builder displays a panel asking if you want to add **willy.tiff** to the project. Click Yes, and Project Builder's window comes forward to show you that the file has been added under Images in the Files browser.

(In general, it's best to add TIFF or EPS format files to a project rather than use them to create local images. By adding the image file to the project you make one copy of the image data available to all nib files in the project. If, on the other hand, you ask Interface Builder to create an image with the data, the image data is copied from the image file into the nib file. Thus, each nib file that requires the image would have to have a separate copy of the data.)

If the image that you add to the File window is no larger than 48 by 48 pixels, the Images display shows the actual image. Larger images (as in this case) are displayed by Interface Builder's Image Inspector.

The Image Inspector 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 16-10 shows a detail of the Image Inspector.

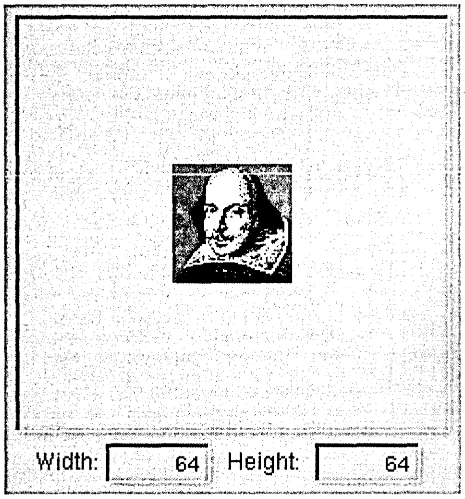


Figure 16-10. The Image Inspector

To place the image on a button in your application, simply drag the icon from the File 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 image isn't transferred.)

Adding Sound

To manipulate the sounds in your application, Interface Builder provides two tools, the Sounds display of the File window and the Sound Inspector. The Sounds display is the repository for your application's sound resources. By dragging a sound icon from the Sounds display onto a Button object in your application, you can associate a sound with that object. The Sound Inspector lets you play 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 display by clicking the Sounds suitcase in the File 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 graphic representation of the selected sound's waveform. The graph plots the change of the sound's amplitude over time. You can play the entire sound 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.

Using your computer's microphone, you can replace the selection in the Sound Inspector with sound you record. Click the Record button to start recording. When you're through recording, click Stop to end the recording session and display the waveform. 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 the File Viewer to the File window. Interface Builder will ask if you want to add the sound file to the project.

Using the File Viewer, 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 Sound Inspector shows the sound's waveform. Click Play to hear the sound.

Now, let's create a sound for the Calculate 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 16-11, makes a satisfying button-click sound.

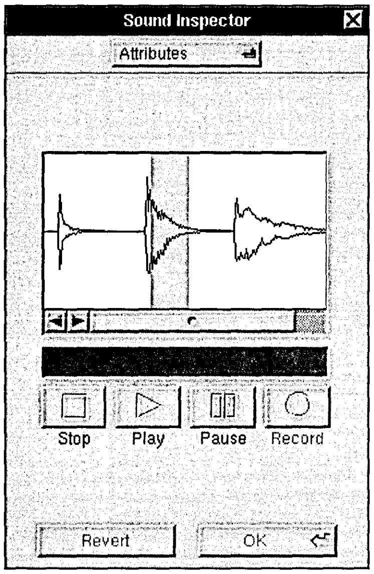


Figure 16-11. The Sound Inspector

Once you've found a portion of the waveform that you want for the Calculate button, select and then delete the portions that precede and follow it. Click OK to save the modified sound.

Let's associate the sound with the Calculate button. Drag the sound icon from the Sounds window to the Calculate 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 Calculate.

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.

17 *Building a Text Editor Using Multiple Nib Files*

17 *Building a Text Editor Using Multiple Nib Files*

Most larger applications benefit from storing different parts of their interface in separate nib files. The primary elements of the interface—the main menu and perhaps a window or two—are contained in one nib file, and the other parts of the interface are contained in one or more auxiliary nib files. When the application starts, its primary interface objects are created immediately. Objects specified in its auxiliary nib files are created only on demand, as when a user requests an Info panel.

This program design is a consequence of the way nib files are accessed by an application. As you've seen in the earlier projects, all objects described in a nib file are created at the same time:

```
[NXApp loadNibSection:"Interface.nib" owner:NXApp];
```

There's no way to load a subset of a nib file's objects. However, the same functionality can be gained by using multiple nib files.

Using multiple nib files can improve your application's perceived performance. If at start-up 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, when users attempt to access other parts of the application, they will experience small delays as new objects are created from the auxiliary nib 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 nib file is to let an application replicate a piece of its interface any number of times. The document windows in Edit provide a good example. Since it can't be predicted how many document windows a user might need, the application must offer a way to create an unlimited number of them. By putting the document window interface in a separate nib file, each time a user requests another window, a new set of objects can be created from the file.

This project demonstrates how to use multiple nib files in an application. Before tackling the more advanced problem of using auxiliary nib files to replicate a piece of an application's interface, let's see how to use such a nib file to store an infrequently accessed user-interface object, the Info panel.

Adding an Info Panel to Your Application

An Info panel is an important component of your application's user interface; however, in practice users rarely access it. By putting the Info panel in a separate nib file, you can reduce your application's start-up time and memory usage. Let's see how this is done.

Close any other projects you may still have open and then choose the New command from Project Builder's Project menu. Save the new project in your home directory under the name "TextEditor". In Project Builder's Files display, locate the entry for the interface file **TextEditor.nib** and double-click it to start Interface Builder. When Interface Builder starts, the new application's main menu and standard window appear. For now, these two components will constitute the application's primary user interface.

Next, let's create a class, the Distributor class, that defines an object to manage the Info panel. A Distributor object will be the target of an action message from the Info menu item. When it receives the Info item's message, the Distributor object will load the Info panel's interface.

To create the Distributor class, switch to the Classes display of Interface Builder's File 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. Using the Class Inspector, change the name of this class to "Distributor".

The next step is to declare the Distributor class's single outlet and action method. Make sure the Outlets button in the Class Inspector is highlighted and then enter **infoPanel** in the text field. Click Add Outlet. Next, click the Actions button and then enter **showInfoPanel:** in the text field. Click Add Action. The Attributes display should now look like this:

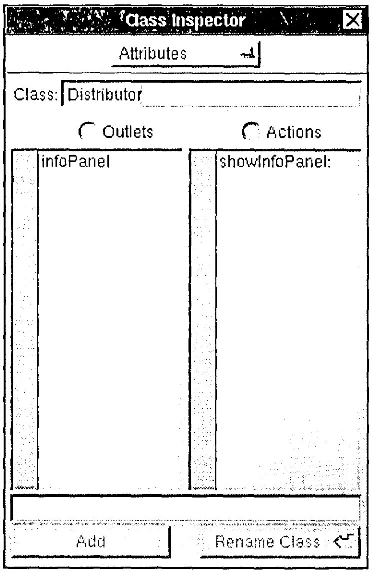


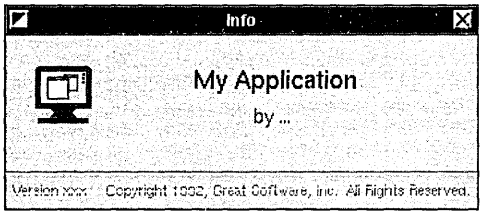
Figure 17-1. Attributes Display for the Distributor Class

To create template source code files for the Distributor class, drag to Unparse in the File window's pull-down list. Two panels open in succession: The first asks you to confirm that you want to create these class files, and the second asks whether these class files should be added to the project. Click OK in each panel. If you look at the Files display in the Project Inspector, you'll notice that **Distributor.h** is listed under Headers and **Distributor.m** is listed under Classes. We'll defer writing the **showInfoPanel:** method until the nib file that contains the Info panel has been created.

Now, let's create an object of the Distributor class and make it the target of the Info command. With the Distributor class selected in the File window, drag to Instantiate in the pull-down list. When you release the mouse button, the File window switches to the Objects display to display the icon for the new custom object. This icon is titled "Distributor".

Control-drag a connection from the Info command in your application's main menu to the Distributor icon. The Inspector panel shows the Connections display for the MenuCell Inspector. Double-click the **showInfoPanel:** action to make the connection. Now, whenever the user chooses the Info command, a **showInfoPanel:** action message will be sent to the Distributor object. The Distributor object will then have to load the auxiliary nib file that contains the Info panel. Save the TextEditor nib file before proceeding.

To build the auxiliary nib file (which is known as a "module"), choose the New Module command from the Document menu. This command opens a submenu of module types. Choose the New Info Panel command. A new File window opens and a template Info panel appears.



My Application
by ...

Version x.x.x Copyright 1992, Great Software, Inc. All Rights Reserved.

Figure 17-2. Info Panel Template

Customize the text in the panel by changing the application name to "TextEditor" and by adding your name to the byline. Save the interface you've created in a file named **Info.nib**, in the same directory that holds **TextEditor.nib**. (For example, if your language preference is set to English, this directory will be **~/TextEditor/English.lproj**). Answer Yes to the panel that asks whether you want to add this nib file to the project.

The auxiliary nib file is complete except for connecting the user interface it provides to a Distributor object, the owner of this interface. Before we can connect these two, we must make the Distributor class known within the Info nib file. (So far, the interface to the Distributor class is known only within the TextEditor nib file, where it was declared.)

To make the interface to the Distributor class known within the Info nib file, Interface Builder must parse the class interface file, **Distributor.h**. Switch to the Classes display in the File window for **Info.nib**. Drag to Parse in the pull-down list. In the Open panel that appears, select **Distributor.h** and click OK. The class appears in its proper place in the File window's class hierarchy, and you can view its interface using the Class Inspector.

Now that the Distributor class is known, you can make a Distributor object the owner of the Info nib file. Switch to the Objects display of the File window and select the File's Owner object. The File's Owner Inspector reveals that the file's owner is an instance of the Object class. To reassign the class of the file's owner, click the Distributor entry in the inspector.

Now, connect the Distributor object (the File's Owner object) in the File window to the Info panel. Control-drag a connection from the file's owner to the title bar of the Info panel. In the Connections display of the File's Owner Inspector, double-click the **infoPanel** outlet to establish the connection.. Save **Info.nib**.

The graphic part of the interface is done; let's write the **showInfoPanel:** method for the Distributor class. Open **Distributor.m**. (You can do this by double-clicking the class's entry in Interface Builder's class hierarchy browser.) In **Distributor.m**, make the changes that are listed in bold below:

```
#import <appkit/appkit.h>
#import "Distributor.h"

@implementation Distributor

- showInfoPanel:sender
{
    if (!infoPanel)
        [NXApp loadNibSection:"Info.nib" owner:self];
    [infoPanel makeKeyAndOrderFront:self];
    return self;
}

@end
```

The **showInfoPanel:** method above checks whether an Info panel has already been created. If not, a new one is unarchived from the **Info.nib** file. As the Info panel and the objects are unarchived from the nib file, the Application Kit initializes the Distributor object's **infoPanel** outlet to the **id** of the new Info panel. Finally, this method sends a message to the Info panel (through the **infoPanel** instance variable) to become the key window and order itself to the front of its window tier.

After you save the **Distributor.m** file, the program is ready to compile and test. Click the Run button in Project Builder's project window. When the application begins running, check the operation of the Info command. Notice that the first time you choose the Info command, there's a slight pause before the Info panel appears. However, if you close the panel and choose the Info command a second time, the panel appears instantly. The first time you summon the panel, it must be unarchived from the nib file; thereafter, the panel is simply being ordered on and off the screen list. (If the Info panel doesn't appear when you choose the Info command, quit the program and recheck the connections in Interface Builder.)

So far, this project has demonstrated how to isolate rarely used interface objects in a nib file of their own. The following sections expand on the program to show how to use a separate nib file as a source of document windows for the text editor. Let's take a look at the design of the text editor.

The Text Editor's Design

The text editor has a simple user interface: Through the application's Document 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:

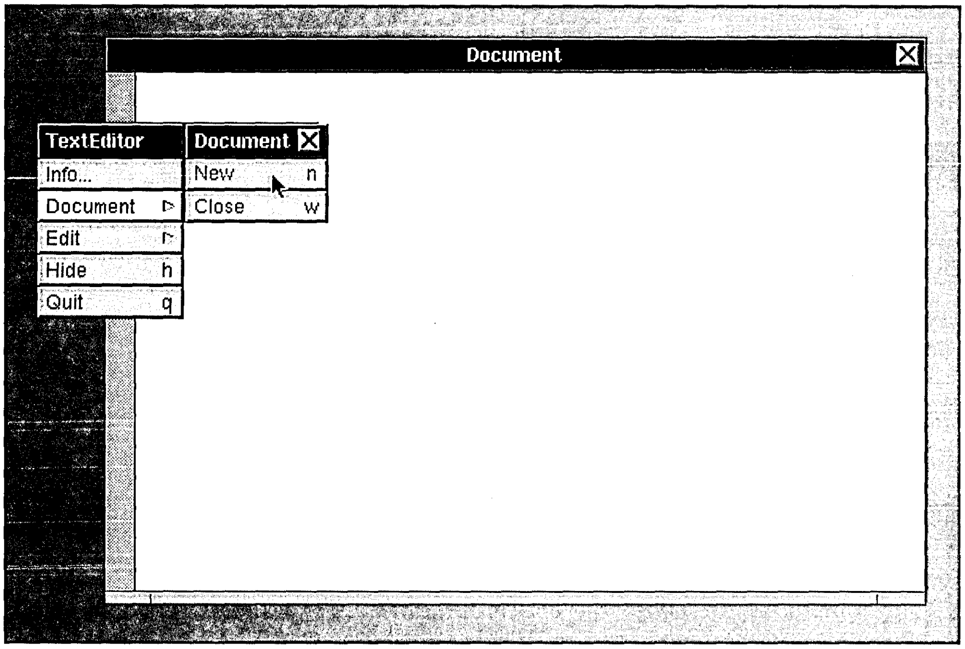


Figure 17-3. The Text Editor

The interface you see in Figure 17-3 is created using two nib files. The main nib file contains the specification for the application's main menu and its submenus. An auxiliary nib file contains the specification for a document window and its scrolling text area. The two interfaces are linked by two custom objects (one of the Distributor class and one of the Document class), as shown in Figure 17-4.

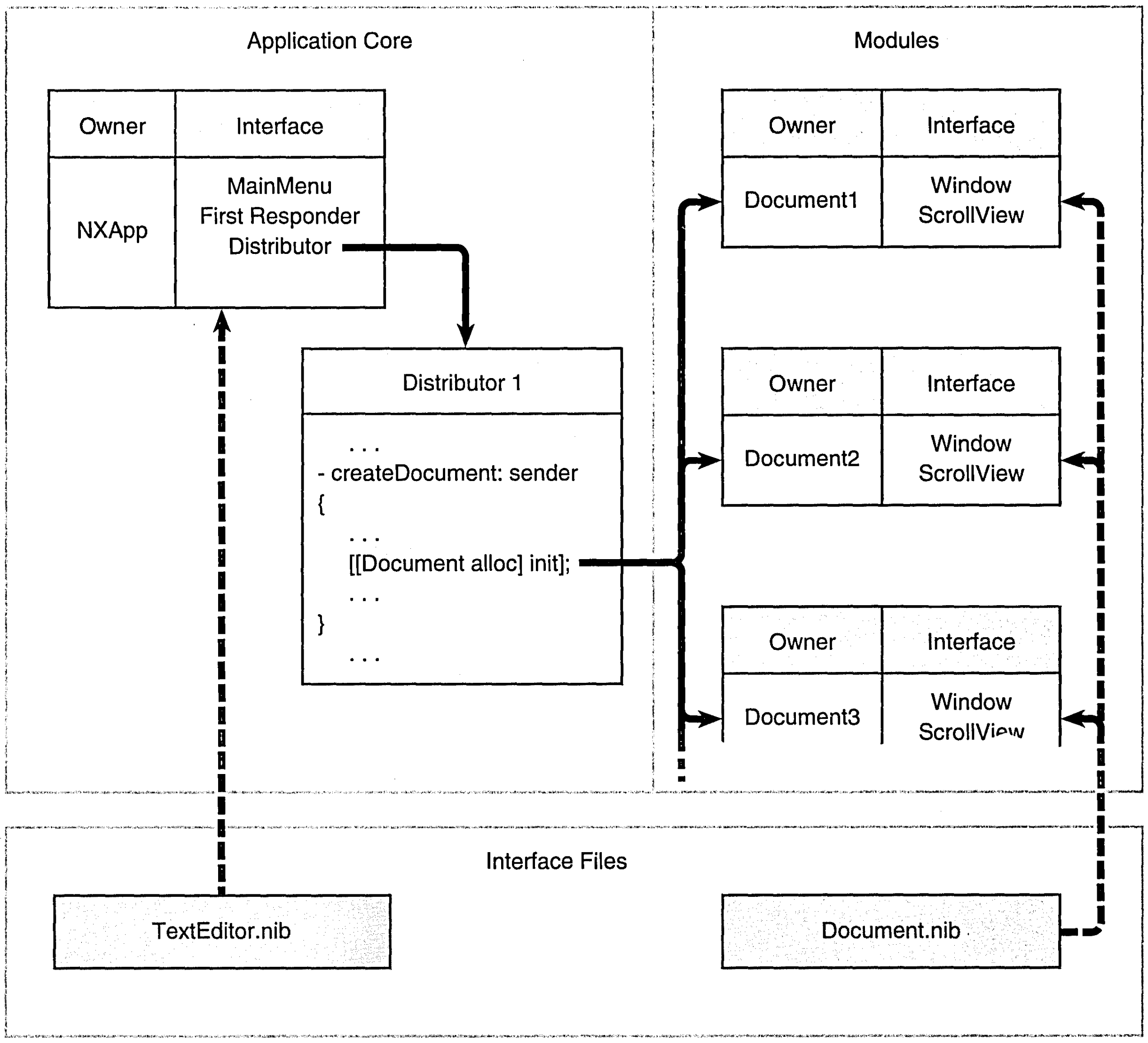


Figure 17-4. The Application's Design

When the application starts, the primary interface objects are created from the specification in **TextEditor.nib** and connected to their owner, NXApp. An object of the Distributor class 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 Document menu, a **createDocument:** action message is sent to the Distributor object. As you can see in the figure, this object in turn creates and initializes a new object of the Document class, a class you'll define in the process of building this application.

The **init** method in the Document class contains these lines:

```
- init
{
    [super init];
    [NXApp loadNibSection:"Document.nib" owner:self];
    return self;
}
```

Each Document object, as it's initialized, is made the owner of a set of objects specified in the **Document.nib** file. Thus, each time the user clicks the New command, a new Document object along with a new window and scrolling text area are created.

The design introduced here is common for applications that replicate pieces of their user interface. 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. In this way, a module can be 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.

In contrast, recall how the Info panel is implemented. With the Info panel, an object within the application's core, the Distributor object, loads the auxiliary nib file. However, the interface module isn't designed to be replicated (in fact, quite the opposite) nor is there any state information that needs to be retained by the module.

Modifying the Application's Interface

Let's implement the design described above by modifying the TextEditor application created so far. First, since the Info nib file is no longer needed, close it by selecting the File window titled "Info.nib" and choosing the Close command in Interface Builder's Document menu.

Next, modify the TextEditor nib file by removing the window object. As explained previously, the application's main nib file doesn't include a document window—document windows are provided by the auxiliary nib 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 Document menu item to your application's main menu. Position this item immediately above Edit. The Document menu that opens displays more commands than you'll need in this project. Cut all but the New and Close commands from the menu.

This completes the visible part of the interface for the application's core. Save the nib file. Next, we'll modify the Distributor class.

Modifying the Distributor Class

The Distributor class must be modified so that new document windows are created whenever a user chooses the New command from the application's Window menu. When a Distributor object creates each new document window module, it temporarily stores the identity of the module's owner. In a more robust application, the Distributor object 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 modify the Distributor class, switch to the Classes display of the File window and select the Distributor class. Next, open the Class Inspector, if it isn't open already. Now, add another action message by clicking the Actions button in the Class Inspector and then entering **createDocument:** in the text field. Click Add Action. Now that the new method has been declared in the nib file, you must add it to the class files.

Editing the Class Files

Double-click the **Distributor** entry in the Files window to open 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**:

```
#import <appkit/appkit.h>

@interface Distributor:Object
{
    id infoPanel;
    id newDocument;
}

- showInfoPanel:sender;
- createDocument:sender;
@end
```

Also, make these changes to the class implementation file, **Distributor.m**:

```
#import "Distributor.h"
#import "Document.h"

@implementation Distributor

- showInfoPanel:sender
{
    if (!infoPanel)
        [NXApp loadNibSection:"Info.nib" owner:self];
    [infoPanel makeKeyAndOrderFront:self];
    return self;
}

- createDocument:sender
{
    newDocument = [[Document alloc] init];
    [newDocument show:self];
    return self;
}

@end
```

Each time a Distributor object receives a **createDocument:** message, it creates a new Document object (the owner of the document window module) and stores the object's **id** in its **newDocument** variable. Next, it sends a **show:** message to the new Document object. As you'll see when you define the Document class, this message brings the module's document window to the front of its tier on the screen and makes it the key window.

As suggested earlier, in a more complex application, the Distributor object might keep track of each Document object it creates so that it can send messages to any one of them. For example, it might use an object of the List or HashTable class to record the **id**s of each of the Document objects it creates.

After you've made these changes to the class files, save them and close the Edit windows.

Connecting the Objects

Now, let's connect the New command to the Distributor object. First, notice that the New command is disabled—its title is in gray. Interface Builder disables menu items from the menu palette that aren't already connected to some target. To enable the New item, select it and switch to the Attributes Inspector. Click the button titled "Disabled" to remove the check mark. The New command is now displayed in black.

Next, in the File window, switch to the Objects display. Control-drag a connection from the New command in your application's Document menu to the Distributor object in the File window. The Inspector panel shows the Connections display for the MenuCell Inspector. Make sure the **target** outlet and the **createDocument:** action are selected and click Connect. Now, whenever the user chooses New, a **createDocument:** action message will be sent to the Distributor object.

This completes the main nib file; 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 **TextEditor.nib** file. (Choose Close from the Document menu.) You can also close the Inspector panel.

Creating the Module's Interface

A module consists of a window, a scrolling text area, and a custom object that owns this interface. The custom object will be of the Document class, a subclass of Object that you'll define shortly.

Choose the New Module command from the Document menu. From the New Module menu that appears, choose New Empty. This command produces a nib file containing only the most basic components. You can see from the File window that appears that this module consists only of an owner object and a First Responder.

Drag a window from the Palettes window into the workspace and open the Window Inspector. Change the title of the window to "Document". Now, drag a ScrollView from the Scrolling Views display of the Palettes window into the document window and resize it so that it covers most of the window's area. Save the nib file you've created so far in a file named **Document.nib**. Also, in the attention panel that appears, confirm that this file should be added to the TextEditor project.

The visible portion of the module's interface is complete; the next job is to define the owner object.

Defining the Document Class

In the File window, switch to the Classes display. Create a new subclass of Object by selecting the Object entry and dragging to the Subclass command in the pull-down list. Using the Class Inspector, name this new class the "Document" class.

The next step is to define the outlets and actions of the Document class. Following the same general steps you took with the Distributor class, give the Document class a **myWindow** outlet and a **show:** action method. Create class definition files for the Document (that is, drag to the Unparse button in the File window) and add these files to the project.

Editing the Class Files

As before, edit the class interface file **Document.h** by adding the line that appears in bold:

```
#import <appkit/appkit.h>

@interface Document:Object
{
    id myWindow;
}

- init;
- show:sender;
@end
```

Also, make these changes to the class implementation file **Document.m**:

```
#import <appkit/appkit.h>
#import "Document.h"

@implementation Document

- init
{
    [super init];
    [NXApp loadNibSection:"Document.nib" owner:self];
    return self;
}

- show:sender
{
    [myWindow makeKeyAndOrderFront:self];
    return self;
}

@end
```

The **init** method initializes a new Document 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 Document's **myWindow** outlet.

Connecting the Objects

A Document object owns the user-interface objects that are unarchived from **Document.nib**. Before you can connect the owner to its interface, you must specify that the owner is of the Document class. Switch to the Objects display in the File window and select the File's Owner object. Using the Inspector panel, click "Document" to assign the class of the owner object.

The owner 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 Document 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 panel's Connections display and click Connect. Finally, save the finished nib file.

Now that 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 the workspace by closing any of Interface Builder's windows you no longer need.

Compiling and Running the Application

Click Run in Project Builder's project window. If the project file needs to be saved, Project Builder asks if you want to save it before proceeding.

When that application begins running, 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. You could also, for example, implement the Close command or the Document commands that you previously deleted, such as Open and Save. Or you might make it so each new Document window opens in a location offset from the previous one so that old windows aren't obscured by new ones.

18 *Building a Custom Palette*

18 *Building a Custom Palette*

As you've seen, Interface Builder gives you convenient access to objects defined in the Application Kit: You can drag objects directly from the Palettes window into your application. Through *custom palettes*, Interface Builder lets you extend this pattern of access to classes of your own design.

A custom palette is a display that can be added to Interface Builder's Palettes window. Each custom palette is represented by its own button at the top of the Palettes window. When the button is clicked, the palette's object or objects appear in the lower portion of the window. Custom palettes can contain subclasses of:

Class	**Comment**
View	Must be dragged into a window
Object	Must be dragged to the File window
Window	Can be dragged and dropped anywhere
MenuCell	Must be dragged into a menu

You can manipulate these objects just as you would the objects on the standard palettes. They can be dragged into the application under construction, resized and relocated through direct mouse actions (if they are View or Window objects), and inspected using the Inspector panel. When Interface Builder is put in test mode, objects from custom palettes are fully functional. For example, View objects draw themselves and react to mouse events just as they would in a real application. (This is in contrast to CustomViews, as described in more detail later in this project.)

Custom palettes let you tailor your development environment to suit your needs. They also provide a convenient way to distribute classes to other developers. It's important to note, however, that only classes that meet the following criteria are good candidates for custom palettes.

- The class should be designed for reusability. That is, it should be easily adapted for use in a wide range of applications. An object that must be the delegate of the Application object, for example, will be difficult to accommodate in many applications.
- The class should define objects that are useful in a variety of applications. There's little advantage in creating a custom palette for an object that will be used infrequently.
- The class should be thoroughly debugged. The best approach to creating a custom palette for a new class is to first debug the class by building test applications and then, when it's debugged, build the custom palette.

This project first describes how to create a simple custom palette and then shows you how to add an inspector for the custom object that the palette contains. Before starting, let's look at how custom palettes fit into the overall structure of Interface Builder.

Custom Palettes and Interface Builder

The previous projects have demonstrated that you build a NeXTSTEP application by designing its interface, defining your own classes as needed, connecting the components, and then compiling the application. This creates an application that consists of executable code and archived data. At run time, some objects are instantiated directly (such as the Application object) and others are unarchived from nib files.

Interface Builder is no different in the way it is constructed. For example, the first time you click the Scrolling Views button in the Palettes window, Interface Builder loads a bundle containing executable code and archived data for the appropriate objects and displays these objects in the Palettes window. (For information on bundles, see the class specification for NXBundle, a common class.)

Adding a new palette to the Palettes window, then, involves creating a type of bundle that Interface Builder can load into itself at run time. This bundle is called a *palette file* and has a ".palette" extension. Palette files contain archived versions of the objects to be displayed in the Palette window and compiled code to support these objects. A palette file can also contain archived data and object modules for the Inspector associated with a custom palette object.

Custom palettes are loaded into Interface Builder dynamically. That is, when a user chooses the Load Palette command from the Tools menu and specifies a palette file, Interface Builder opens the palette file, loads the object modules it contains and then unarchives the objects that will appear in the Palette window. Thereafter, the classes of these custom objects are known to Interface Builder: Their names appear in the proper places in the Classes window, their outlets and actions appear in the Connections Inspector, and Interface Builder can create new objects of these classes as needed. Using Interface Builder's Preferences panel you can have one or more custom palettes loaded automatically whenever Interface Builder is launched.

The Custom Object's Design

The palette we'll create in this project contains a single custom object, a ProgressView. A ProgressView reflects the progress of a long-running operation by filling with dark gray an ever increasing proportion of its horizontal extent:

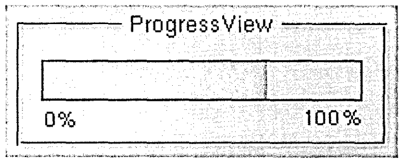


Figure 18-1. ProgressView, Box, and TextField Objects

You could use such an object to inform the user of the status of a long-running calculation, file operation, or other process. A ProgressView responds to an **increment:** message by increasing the length of the gray bar a predetermined amount. We'll take a closer look at the implementation of the ProgressView class after creating the interface for the custom palette.

Creating the Interface

The primary component of a custom palette project is a nib file. This file contains the archived object (or objects) that will appear in the palette and a TIFF image that will be used for the button at the top of the Palettes window.

To begin the palette project, start Project Builder and, from the Project menu, choose New. In the panel that appears, drag to Palette in the Project Type pop-up list. Give the project the name "ProgressPalette" and save it in your home directory. The Project window for this project appears.

If you browse the Files display of the Project window, you'll find that Project Builder has added these files:

File	**Description**
ProgressPalette.[hm]	Subclass of IBPalette (which is declared in **/NextDeveloper/Headers/Apps**). For palettes that contain only View objects, nothing must be done to these files. For other types of palettes, one or more methods must be implemented.
ProgressPalette.nib	Interface archive for the palette project. You'll use Interface Builder to modify this template file by adding the objects that will be part of your palette.
palette.table	Loading instructions for Interface Builder. This file becomes part of the palette file package. It tells Interface Builder which icon to display for the palette and which classes to add to the Classes display of the File window, among other things.
Makefile	Standard makefile for palette projects. Project Builder maintains this file; you shouldn't change its contents directly.

Double-click the **ProgressPalette.nib** entry. Interface Builder starts and displays the contents of this template file: a File's Owner object, the first responder, and a window titled "Palette View". Using Interface Builder's Inspector window, you can verify that the File's Owner object in the File window is an object of the ProgressPalette class. If you switch to the Connections display, you'll see that the **originalWindow** outlet, which the File's Owner inherits from its parent class, is already attached to the panel. When Interface Builder loads a palette file, it uses this connection to find the View objects that it must extract from the nib file and position within the Palette window.

The next step is to put a ProgressView object in this panel; however, at this point the ProgressView class is unknown to Interface Builder. Interface Builder provides a generic View, the CustomView object, for these situations. A CustomView object records the location, width, height, and class of a View of your own design. At run time, when objects are unarchived from the nib file, an object of the class you specified is created in place of the CustomView. The View that's created is positioned and resized to match the position and dimensions of the CustomView in the nib file, and its **drawSelf::** method is invoked to cause it to display itself.

Drag a CustomView (from the Basic Views palette) into the panel and resize it to look like this:

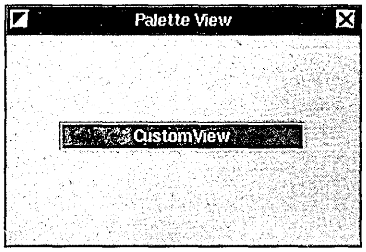


Figure 18-2. Building the Custom Palette

Now add a label to this CustomView by dragging a TextField titled "Title" from the Palette window into the panel. Change the TextField to read "0%" and use the Font panel to decrease the font size. Make a second label by copying and pasting the one you've just created and then change the title to read "100%". Position these titles at opposite ends of the CustomView. Finally, create a box around the CustomView and the titles by selecting the three objects and choosing the Group command from the Layout menu. Change the title of the box to read "ProgressView".

Defining the ProgressView Class

The next step is to reassign the class of the CustomView to the as-yet-unwritten ProgressView class. To do this, you must first use the Classes display of the File window to define the ProgressView class.

Switch to the Classes display and select View in the class hierarchy. Create a subclass of View by dragging to Subclass in the Operations pull-down list. A new class titled "MyView" appears in the hierarchy. Using the Class inspector, change the name of this class to "ProgressView" and press Return.

A ProgressView responds to a single action method: **increment:**. Using the Class inspector, add this method to those listed under the Actions button.

Now, let's create template source files for the ProgressView class. Drag to Unparse in the Operations pull-down list in the File window. In the first panel that appears, confirm that you want to create **ProgressView.h** and **ProgressView.m**. In the second panel, confirm that you want these files added to the project. We'll fill in these template files later.

Finally, reassign the class of the CustomView in the panel. Select the CustomView. Note that the CustomView is grouped within a Box object (that is, it's within the Box's view hierarchy). Clicking the box selects the Box object. To move the focus of selection to the objects inside the box, double-click within the box. Now select the CustomView by clicking it. Using the Attributes display of the CustomView Inspector, locate ProgressView in the list of classes it contains. Click this entry, and the label in the CustomView changes to "ProgressView".

Providing An Image for the Palette's Button

The ProgressView custom palette needs an image for its button in the Palettes window. You can either use the IconBuilder application (in **/NextDeveloper/Apps**) to create a new one, or you can use **ProgressPalette.tiff**, which you'll find in **/NextLibrary/Documentation/NextDev/Examples/IBTutorial/Images**. (If you create your own image, make sure it's no larger than 48 by 48 pixels and that its background is transparent.)

Drag the image's file icon from the Workspace Manager File Viewer into Project Builder's Project window. As you drag the icon into the window, a suitcase opens to accept it. When you release the icon, the image is added under Images in the Files display. Now save the project. When the project is saved, Interface Builder is alerted to any changes it contains. Now if you look in the Images display of Interface Builder's File window, you'll see the new image.

The custom palette's interface is complete. The next step is to write the class definition files for the ProgressView class. Before continuing to the next step, save the nib file you've created.

Writing the ProgressView Class Files

The files that Interface Builder has created for the ProgressView class contain only template code for the **increment:** action method. You'll have to finish this method's implementation and add the other methods described in this section. (If you're reading an electronic version of this tutorial, you can simply copy the code listed below and paste it into the appropriate ProgressView source file in your project.)

ProgressView.h

This file declares the interface to the ProgressView class.

```
#import <appkit/appkit.h>
#define DEFAULTSTEPSIZE 5
#define MAXSIZE 100

@interface ProgressView:View
{
    int total, count, stepSize;
    float ratio;
}
- initFrame:(const NXRect *)frameRect;
- drawSelf:(const NXRect *)rects :(int)rectCount;
- setStepSize:(int)value;
- (int)stepSize;
- setRatio:(float)newRatio;
- increment:sender;
- read:(NXTypedStream*)stream;
- write:(NXTypedStream*)stream;

@end
```

The file starts by importing the standard header file for the Application Kit. Then, two constants are defined. DEFAULTSTEPSIZE is the value that's added to a running total each time an **increment:** message is received. MAXSIZE is the maximum length of the bar.

Next, the ProgressView class declares four instance variables:

Variable	**Description**
total	Total length of bar; MAXSIZE in this example
count	Running total; incremented by each **increment:** message
stepSize	Amount to add to **count** upon receiving an **increment:** message
ratio	Proportional length of dark gray portion of bar (**count/total**)

Finally, ProgressView's methods are declared. These methods are discussed in the next section.

ProgressView.m

This file contains the implementation of the methods declared in **ProgressView.h**.

```
#import "ProgressView.h"

@implementation ProgressView

- initFrame:(const NXRect *)frameRect
{
    [super initFrame:frameRect];
    total = MAXSIZE;
    stepSize = DEFAULTSTEPSIZE;
    return self;
}

- drawSelf:(const NXRect *)rects :(int)rectCount
{
    PSsetgray(NX_LTGRAY);
    NXRectFill(&bounds);
    if (ratio > 0) {
        NXRect r = bounds;
        r.size.width = bounds.size.width * ratio;
        PSsetgray(NX_DKGRAY);
        NXRectFill(&r);
    }
    PSsetgray(NX_BLACK);
    NXFrameRect(&bounds);
    return self;
}
```

```
- setStepSize:(int)value
{
    stepSize = value;
    return self;
}

- (int)stepSize
{
    return stepSize;
}

- setRatio:(float)newRatio
{
    if (newRatio > 1.0) newRatio = 1.0;
    if (ratio != newRatio) {
        ratio = newRatio;
        [self display];
    }
    return self;
}

- increment:sender
{
    count += stepSize;
    [self setRatio:(float)count/(float)total];
    return self;
}

- read:(NXTypedStream*)stream
{
    [super read:stream];
    NXReadTypes(stream, "ii", &total, &stepSize);
    return self;
}

- write:(NXTypedStream*)stream
{
    [super write:stream];
    NXWriteTypes(stream, "ii", &total, &stepSize);
    return self;
}

@end
```

ProgressView.m starts by importing **ProgressView.h** for the interface to its own class. The rest of the file contains the implementation of ProgressView's methods:

Method	**Description**
initFrame:	Initializes a newly allocated ProgressView by setting the values of its **total** and **stepSize** variables. Its **count** and **ratio** instance variables are automatically initialized to 0.
drawSelf::	Draws the ProgressView by first filling its entire bounds rectangle with light gray, determining which portion of the bounds should be filled with dark gray and painting that portion, and finally drawing a black border around the entire ProgressView.
setStepSize:	Sets the amount to be added to **count** when the ProgressView receives an **increment:** message. (This method will be used in the next project.)
stepSize:	Returns the amount to be added to **count** when the ProgressView receives an **increment:** message. (This method will be used in the next project.)
setRatio:	Sets the **ratio** variable and then redisplays the ProgressView (thus causing the **drawSelf::** method to be invoked).
increment:	Increases the value of the **count** variable by adding **stepSize** to it. This method then invokes the **setRatio:** method, using the ratio of **count** to **total** as the argument.
read:	Reads the archived values of the **total** and **stepSize** variables from a typed stream.
write:	Writes the values of the **total** and **stepSize** variables to a typed stream.

At a minimum, a custom palette object must be able to draw, archive, and unarchive itself; thus, the **drawSelf::**, **write:**, and **read:** methods above. The other methods are peculiar to the ProgressView class and aren't required by all custom palette objects.

The **drawSelf::** method is invoked when the palette is first loaded, to draw the ProgressView in the Palettes window. It's also invoked when you put Interface Builder in test mode and there's a ProgressView object in your application's window. Of course, when an application that contains a ProgressView is run, **drawSelf::** is invoked whenever the ProgressView needs to draw itself, such as after it receives an **increment:** message.

The **read:** and **write:** methods are needed so that the ProgressView can be archived. When you create the custom palette, the ProgressView object must archive itself into the palette file. When the custom palette is loaded into Interface Builder, the ProgressView is unarchived from the palette file.

The process of unarchiving involves allocating enough memory for the object and then sending it a **read:** message so that it can initialize its variables from the values stored in the archive. In unarchiving, the **initFrame:** method, which would normally establish the values of the **total** and **stepSize** variables, isn't invoked. Thus, the **read:** method must establish those values. The matching **write:** method records the values of **total** and **stepSize** in the archive file when the palette is created. Without these methods, a newly unarchived ProgressView would have 0 as the values of **total** and **stepSize**. Whenever you create a class that you intend to use in a custom palette, remember to implement **read:** and **write:** methods to archive the variables whose values you want to store along with the object.

Updating the palette.table File

Before you can compile the palette project, you must update the **palette.table** file. Interface Builder consults this table when it loads a palette file. It uses the information from the table to identify and instantiate the nib file's owner, to display the proper image for the button in the Palette window, and to insert class names within the Classes display of the File window, among other things.

Locate **palette.table** under Other Resources in Project Builder's Project window. Double-click the entry to reveal the file's contents:

```
Class = ProgressPalette;          /* (a subclass of IBPalette) */
NibFile = ProgressPalette;        /* (a nib file name) */
/* Icon =;                           (a tiff/eps file name) */
/* ExportClasses = ();               (a list of class names) */
/* ExportImages = ();                (a list of icon names) */
/* ExportSounds = ();                (a list of sound names) */
```

The first two lines identify the names of the class of the nib file's owner and of the nib file itself. The remaining lines can be used to identify other elements of the palette file. For this project, you need to specify the name of the image to be used for the Palette window button and to specify the name of the class, ProgressView, that should be added to the Classes browser. Make the changes that appear in bold below:

```
Class = ProgressPalette;          /* (a subclass of IBPalette) */
NibFile = ProgressPalette;        /* (a nib file name) */
Icon = ProgressPalette;           /* (a tiff/eps file name) */
ExportClasses = (ProgressView);   /* (a list of class names) */
/* ExportImages = ();                (a list of icon names) */
/* ExportSounds = ();                (a list of sound names) */
```

After you've made these changes, save and close **palette.table**.

Compiling and Loading the Palette

You're now ready to compile the custom palette. Switch to Project Builder's Builder display and click the Build button. When the process is finished, a file with the name **ProgressPalette.palette** is added to the project directory.

To load the custom palette, choose Interface Builder's Load Palette command from its Tools menu. In the Open panel that appears, select that palette file and click OK. After a moment, Interface Builder's Palette window is updated to show the new palette.

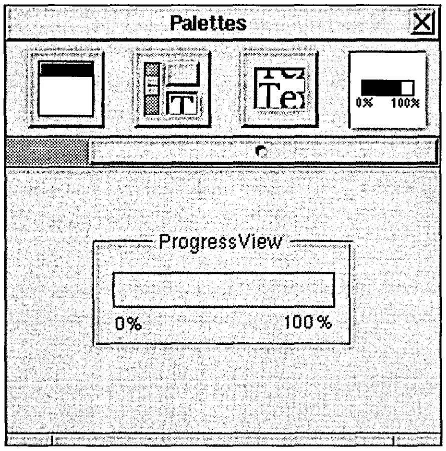


Figure 18-3. ProgressView Custom Palette

Notice that a horizontal scroller appears to give you access to palette buttons that no longer fit within the Palettes window.

Testing the Palette

Now that a ProgressView is available from within Interface Builder, it's easy to test its operation. Close the palette project by closing the Project window for the ProgressPalette project. Now, start a new project by choosing New in Project Builder's Project menu. In the panel that appears, name the project "Test" and make sure the Project Type button reads "Application". Finally, open the nib file.

In Interface Builder, drag a ProgressView object from the Palettes window into the application's window. To test the ProgressView's operation, you have to send it **increment:** action messages. Add a button to the window and change its title to "Increment". Control-drag a connection from the button to the ProgressView. (Make sure the connection is made to the ProgressView and not to the Box that surrounds it—check the Connections list in the Connections Inspector to confirm the identity of the destination object.) Using the Connections Inspector, specify that the Button sends an **increment:** message to the ProgressView.

Finally, test the ProgressView by putting Interface Builder in test mode and clicking the Increment button. The gray bar should step across the ProgressView with each click. If nothing happens, quit the test mode, recheck the connection between the button and the ProgressView, and try again.

Using Custom Palette Objects in Other Applications

Building an application using a custom palette object is in most respects identical to building one using the standard objects that are available within Interface Builder; the few differences are described here.

You've demonstrated that the ProgressView custom object works within Interface Builder's test mode. However, if you compile the new application and attempt to run it, this error appears in the Workspace Manager Console window:

```
> objc: class `ProgressView' not linked into application
> An uncaught exception was raised
> Typed streams library error: class error for 'ProgressView': class
not loaded
```

The problem is that although the application's nib file contains an archived ProgressView object, the ProgressView class hasn't been linked into the application. Thus, none of the ProgressView's methods can be invoked.

There are several ways to ensure that the ProgressView class is linked into an application. The easiest is to add the ProgressView class files (**ProgressView.h** and **ProgressView.m**) to the list of class files in Project Builder's Files window. For your own applications, this is a reasonable solution. If, however, you don't want to distribute source code along with the custom palettes you develop, you can either distribute object files (in this case, **ProgressView.o**) or a library containing object modules for your custom classes. The object files or library can be added to the appropriate list in Project Builder.

Another consideration when developing applications using custom palettes concerns the editing of nib files. If you create a nib file that contains a custom palette object, that interface file can be opened only by a similarly configured Interface Builder application. In other words, if the nib file contains a ProgressView, then you will have to load the ProgressView palette before you'll be able to open the nib file. As normally configured, Interface Builder won't have access to the class information for the custom object.

Adding a ProgressView Inspector

A palette file can provide Attribute, Connection, Size, and Help inspectors for the custom objects it contains. (Custom Connection and Help inspectors are rarely needed, however.) For example, when a user attempts to display the Attributes inspector associated with the custom object (say, by selecting the object and choosing Inspector from the Tools menu), Interface Builder loads the inspector and uses it as the Attributes display of the Inspector window. For example, an Attributes Inspector for the ProgressView class might look like this:

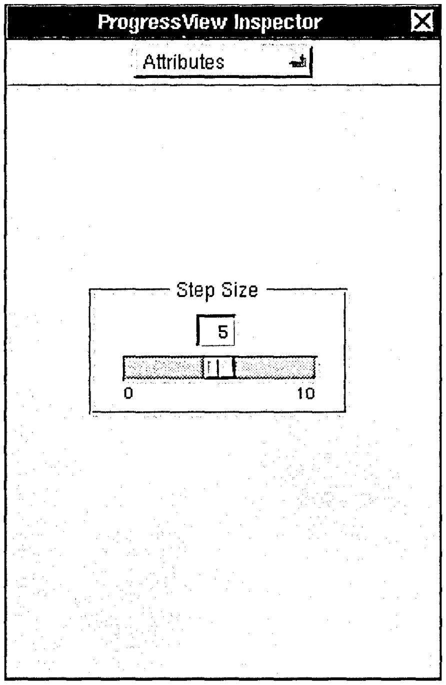


Figure 18-4. ProgressView Inspector

An Attributes Inspector typically lets a user set an object's characteristics that can't be set through direct mouse manipulation. For example, the ProgressView Inspector pictured above lets the user adjust a ProgressView's **stepSize** variable, thus determining the amount the dark gray bar advances across the ProgressView with each **increment:** message.

Inspectors can have OK and Revert buttons, although they aren't required. If the user adjusts the controls in an inspector and then clicks Revert, the changes are discarded and the previous values are reestablished; if the user clicks OK, the new values are sent to the object that's being inspected.

Interface Builder identifies the appropriate inspector to display for a selected object by sending the object one of these messages, depending on the setting of the pop-up list in the Inspector window:

```
getInspectorClassName
getConnectInspectorClassName
getSizeInspectorClassName
getHelpInspectorClassName
```

The **getInspectorClassName** message is sent to determine the name of the class of the Attributes inspector. For example, the ProgressView class could implement this method this way:

```
- (const char *)getInspectorClassName
{
    return "ProgressViewInspector";
}
```

Since each custom object can identify its inspector, a custom palette file can contain multiple classes of objects, each with its own inspector.

Another benefit of this system of identifying an object's inspector is that inspectors are inherited. Interface Builder provides inspectors for each of the classes represented in the Palettes window. If you create a subclass of one of these classes and don't implement the **inspectorName:** method, Interface Builder will display the superclass's inspector whenever the custom object is inspected.

For debugging purposes, it's often better to create the inspector for an object only after the object itself has been debugged and placed in a custom palette. This is the approach we take in this project. Now that ProgressView objects are available through a custom palette, we'll create a simple inspector for the ProgressView class.

Designing the ProgressView Inspector

Creating an inspector for a custom palette object is much like creating the custom palette itself. You start by assembling an interface for the custom object inspector. The owner of this nib file is an object you define that translates actions taken on this interface into messages to send to the object that's being inspected. The class files for the owner object and the inspector's nib file are added to the palette project and compiled into the palette file along with the custom palette object. Let's start by assembling the inspector's user interface.

Interface Builder provides a New Inspector command for our purposes. Choose the New Module command in the Document menu and, in the menu that appears, choose New Inspector. A new File window and a panel titled "Inspector" appear.

Now, let's add some objects to the panel. Drag a horizontal slider into the panel and then add labels for each end (as in Figure 18-4 above). Edit the left label to read "0" and the right one "10". Calibrate the slider to these values by using the Slider Inspector to set its minimum value to 0 and the maximum value to 10. Set the current value to 5 and click OK in the Slider Inspector.

Add an editable text field above the center of the slider. This text field will read out the slider's current setting. Resize the text field to accommodate two-digit numbers and then edit its contents to read "5". Using the Alignment group of buttons in the TextField Inspector, specify that the TextField's display is right aligned:

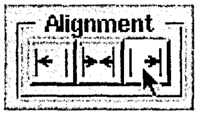


Figure 18-5. Setting the Alignment of the TextField

Finally, select all the objects in the panel and choose the Group command from the Layout menu to surround them in a box. Edit the box's title to read "Step Size".

The interface to the ProgressView inspector is complete. Save the interface in a file called **ProgressViewInspector.nib**—the Open panel will suggest saving the nib file in the proper language directory of the ProgressPalette project—and, when the attention panel appears, confirm that you want the file added to the project.

Designing the ProgressViewInspector Class

The interface that you just created will act on a selected object through the intervention of a ProgressViewInspector object, which we will now define.

Inspectors inherit from Interface Builder's IBInspector class, a subclass of Object. (See **/NextDeveloper/Headers/apps/InterfaceBuilder.h** for the class interface to the IBInspector class.) The IBInspector class has these important outlets:

Outlet	**Description**
object	**id** of the object that's being inspected
window	**id** of the Panel that contains the inspector's user interface
okButton	**id** of the OK button, if present
revertButton	**id** of the Revert button, if present

The IBInspector class adopts the IBInspectors protocol, which declares these methods:

Method	**Description**
ok:	Sets the inspected object to reflect the user's choices in the Inspector panel.
revert:	Cancels any pending changes to the inspected object. This method is also invoked when the Inspector is first instantiated.
wantsButtons:	Invoked by Interface Builder to determine if OK and Revert buttons should be displayed for this inspector.

Let's create a subclass of IBInspector for our inspector. The IBInspector class is listed under Object in Interface Builder's Classes browser. Select this entry and drag to Subclass in the Operations pull-down list. In the Class inspector, rename this new class "ProgressViewInspector". Note that the Class Inspector reports that the ProgressViewInspector class inherits the **window** outlet and **ok:** and **revert:** methods of its superclass. A ProgressViewInspector needs two more outlets, which it will use to communicate with the slider and text field in its user interface. Add these outlets and name them **theSlider** and **theTextField**.

Now, create template source files for the ProgressViewInspector class. Drag to Unparse in the Operations pull-down list of the Classes display. In the first panel that appears, confirm that you want to create **ProgressViewInspector.h** and **ProgressViewInspector.m**. In the second panel, confirm that you want to add these files to the project. We'll fill in these template files later.

Next, reassign the class of the File's Owner object to the ProgressViewInspector class. Select the File's Owner object in the Objects display of the File window. In the File's Owner Inspector panel, select ProgressViewInspector. Finally, save the nib file.

Connecting the Objects

The File's Owner, a ProgressViewInspector, must be connected to its user interface objects. Control-drag a connection from the File's Owner to the Slider and connect the two using the **theSlider** outlet. Similarly, connect the File's Owner to the TextField using the **theTextField** outlet. Notice that Interface Builder has already connected the File's Owner and the inspector panel using the **window** outlet. All inspectors are connected to their user interfaces through this outlet.

The Slider and the TextField must also be connected to the File's Owner so that actions taken on these controls are reflected in the object being inspected. Control-drag a connection from the Slider to the File's Owner, and using the Connections inspector, make the File's Owner the target of an **ok:** message from the Slider. Similarly, make the TextField send an **ok:** message to its target, the File's Owner.

These are the only connections you need to make. When the inspector is being used in Interface Builder, its **object** outlet will be set automatically to the **id** of the object that's currently being inspected. When the user clicks OK or Revert in an inspector that has these buttons, Interface Builder will send the appropriate message to the ProgressViewInspector object.

Editing the ProgressViewInspector Class Files

The next step is to fill in the template class files that Interface Builder has added to the project. The finished files are listed here. (If you're reading this on-line, you can copy the listings into the template files in your project.) A description of the files follows each listing.

ProgressViewInspector.h

This file declares the interface to the ProgressViewInspector class.

```
#import <apps/InterfaceBuilder.h>

@interface ProgressViewInspector:IBInspector <IBInspectors>
{
    id    theSlider;
    id    theTextField;
}
- init;

@end
```

The file starts by importing **InterfaceBuilder.h**, which contains the declaration of the IBInspector class, ProgressViewInspector's superclass. Note that the ProgressViewInspector class adopts the IBInspectors protocol, which declares the **ok:**, **revert:** and **wantsButtons:** methods.

This interface file then declares two instance variables, **theSlider** and **theTextField**. These variables correspond to the two outlets that you added using Interface Builder's Class Inspector. Finally, the file declares the **init** method, which is described in the next section.

ProgressViewInspector.m

This file contains the implementation of the methods declared in **ProgressViewInspector.h**.

```
#import "ProgressViewInspector.h"
#import "ProgressView.h"

@implementation ProgressViewInspector
```

```
- init
{
    char buf[MAXPATHLEN + 1];
    id bundle;

    [super init];

    bundle = [NXBundle bundleForClass:[ProgressView class]];
    [bundle getPath:buf
            forResource:"ProgressViewInspector"
            ofType:"nib"];
    [NXApp loadNibFile:buf
           owner:self
           withNames:NO
           fromZone:[self zone]];
    return self;
}

- ok:sender
{
    if (sender == theSlider) {
        [object setStepSize:[theSlider intValue]];
        [theTextField setIntValue:[theSlider intValue]];
    }
    else if (sender == theTextField) {
        [object setStepSize:[theTextField intValue]];
        [theSlider setIntValue:[theTextField intValue]];
    }
    return [super ok:sender];
}

- revert:sender
{
    int step;

    step = [object stepSize];
    [theSlider setIntValue:step];
    [theTextField setIntValue:step];
    return [super revert:sender];
}

- (BOOL)wantsButtons
{
    return NO;
}
```

The **init** method initializes a newly allocated ProgressViewInspector. As in all **init...** methods, the chain of initializations is maintained through a message to the superclass (IBInspector) to initialize itself.

Next, the inspector's user interface is loaded from the appropriate nib file; however, since the palette file could be located anywhere, you have to enlist the services of an NXBundle object to find the proper directories to search. The NXBundle object is initialized on the directory that provided the code for the ProgressViewInspector class. Given the user's language preferences, the nib file will be sought in one of its subdirectories (for example, **English.lproj**, **French.lproj**, etc.). In contrast, if you were to try to load the nib file by sending a **loadNibFile:...** message, Interface Builder's file package would be searched for the nib file. (See the NXBundle class specification for more information.)

When the nib file is loaded, the inspector's **theSlider** and **theTextField** outlets are automatically initialized to the **id**'s of the appropriate objects from the nib file. (In addition, the ProgressViewInspector's **object** outlet is set to the **id** of the ProgressView that the user has selected.)

The **ok:** and **revert:** methods synchronize the values in the inspector panel with each other and with the object being inspected. When a user acts on the slider, for example, an **ok:** message is sent to the ProgressViewInspector. The inspected object's step size is set to the value of the Slider object, and then the TextField's value is made to match that of the Slider.

The **revert:** method asks the selected ProgressView for its current step size and then sends messages to the Slider and TextField to reflect this value. The implementation of each method ends by invoking the IBInspector class's implementation of the identical method:

```
return [super ok:sender];
return [super revert:sender];
```

In the **ok:** and **revert:** methods of inspector classes you write, remember to invoke the IBInspector class's **ok:** and **revert:** methods, as demonstrated here. This is required for the correct operation of inspectors.

Besides being sent when the user clicks Revert, a **revert:** message is sent to the ProgressViewInspector whenever the user selects a ProgressView and the inspector is open. This lets the inspector update its controls so that they reflect the state of the inspected object.

Interface Builder sends a **wantsButtons:** message to the ProgressViewInspector to determine if OK and Revert buttons should appear in the Inspector panel. Most Inspectors won't need these buttons; rather, a user's actions in the panel will immediately and visibly change the state of the inspected object, as in this example.

Modifying the ProgressView Class Files

As mentioned earlier, Interface Builder identifies the inspector for a selected object by sending the object an **inspectorName** message. Since you've created an inspector for ProgressView objects, it's time to add this method to the ProgressView class files.

In **ProgressView.h**, add this declaration:

```
- (const char *)getInspectorClassName;
```

In **ProgressView.m**, add the implementation of the inspectorName method:

```
- (const char *)getInspectorClassName
{
    return "ProgressViewInspector";
}
```

Compiling and Testing the Inspector

After saving the class and nib files, use Project Builder's Build command to compile the project. When the process is finished, you can load the new palette file, **ProgressPalette.palette**. (Remember that only one version of a given palette can be loaded at a time. If you already have an older version of the ProgressView palette loaded, you'll have to restart Interface Builder in order to load the new version.)

Once the custom palette is loaded, test its operation by creating a new application that contains a ProgressView and a button, as illustrated here:

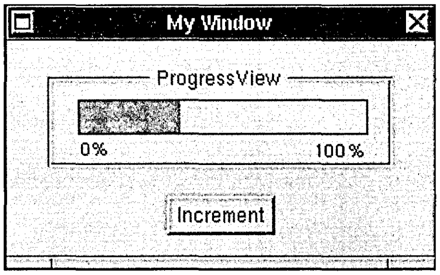
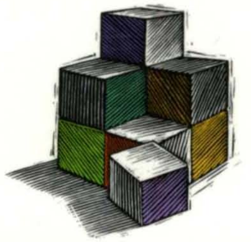


Figure 18-6. Testing the ProgressView Inspector

Use the ProgressView Inspector to set the step size, and then put the application in test mode to test the ProgressView's operation.

If the inspector doesn't appear when you attempt to inspect a ProgressView, recheck the connections in the ProgressViewInspector nib file. (Especially check that the **window** outlet is connected to the panel that contains the inspector's user interface).

Index



NEXTSTEP DEVELOPMENT TOOLS AND TECHNIQUES: RELEASE 3

NeXTSTEP is the object-oriented programming environment that speeds the development of all kinds of software—from mission-critical custom applications for business to advanced research projects for academia. NeXTSTEP offers building blocks that implement essential behavior in a variety of application areas—including database management, telecommunications and networking, and high-quality 2D and 3D graphics.

NeXTSTEP Development Tools and Techniques provides an overview of the application development process and describes the essential tools found in the NeXTSTEP development environment:

- Project Builder and Interface Builder
- The NeXT compiler, preprocessor, and debugger
- Edit, the NeXT mouse-based editor
- Icon Builder, the application icon editor
- Program analysis tools
- Yap, the interactive PostScript previewer
- Terminal, the NeXT VT100 terminal emulator
- Mach object files

NEXTSTEP
Object Oriented Software

53095
9 780201 632491
ISBN 0-201-63249-7
US $30.95
CANADA $39.95

Addison-Wesley Publishing Company