Development Tools

NeXT Developer's Library

NeXTstep

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

Concepts

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

Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

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

Concepts

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

Reference

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

NeXT Development Tools

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

NeXT Operating System Software

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

Writing Loadable Kernel Servers

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

NeXT Technical Summaries

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

Supplemental Documentation

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

NeXT Development Tools

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

Copyright ©1990 by NeXT Computer, Inc. All Rights Reserved.
[2912.00]

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

Notice to U.S. Government End Users:

Restricted Rights Legends

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

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

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

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

Manual written by Gary Miller and Jim Inscore
Edited by Caroline Rose, Roy West, and Helen Casabona
Book design by Eddie Lee
Illustrations by Jeff Yaksick and Don Donoughe
Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs
Publications management by Cathy Novak

Reorder Product #N6007B

Contents

Introduction

1-1 Chapter 1: Putting Together a NeXT Application
1-3 The Application Development Process
1-6 Components of an Application Project
1-12 Putting the Components Together
1-18 Debugging
1-27 Installing an Application

2-1 Chapter 2: The VT100 Terminal Emulator: Terminal
2-4 Introduction to Terminal
2-4 Starting Terminal
2-4 Setting Preferences
2-9 The Main Menu
2-11 The Shell Menu
2-12 The Edit Menu
2-13 The Format Menu
2-14 The Find Menu

3-1 Chapter 3: The NeXT Mouse-Based Editor: Edit
3-3 Starting Up Edit
3-4 Opening Edit Files
3-5 Edit Windows
3-6 Selecting Text
3-6 Using the Ruler
3-9 Contracting and Expanding Text in a File Window
3-11 Setting Preferences
3-17 Keyboard Editing Commands
3-18 Edit and UNIX
3-20 The Main Menu
3-20 The File Menu
3-23 The Edit Menu
3-28 The Find Menu
3-31 The Format Menu
3-32 The Font Menu
3-32 The Text Menu
3-34 The Structure Menu
3-34 The Utilities Menu

4-1 Chapter 4: Developer Applications and Utilities
4-3 The Object Browser Application: AppInspector
4-15 The Malloc Debugger Application: MallocDebug
4-20 The Process Monitoring Application: ProcessMonitor
4-27 NeXT's PostScript Window Server Interface: pft

5-1 Chapter 5: The GNU C Compiler
5-3 GNU CC Command Options
5-14 C Programming Notes
5-17 Legal Considerations

6-1 Chapter 6: The GNU C Preprocessor
6-4 Global Transformations
6-4 Preprocessor Commands
6-5 Header Files
6-6 Macros
6-22 Conditionals
6-27 Pragmas
6-27 Combining Source Files
6-28 C Preprocessor Output
6-28 Invoking the C Preprocessor

7-1 Chapter 7: The GNU Source-Level Debugger
7-5 Summary of GDB
7-6 Compiling Your Program for Debugging
7-7 Running GDB
7-13 Startup Files
7-13 GDB Commands for Specifying Files
7-14 Running Your Program under GDB
7-17 Stopping and Continuing
7-27 Examining the Stack
7-29 Examining Source Files
7-33 Examining Data
7-41 Examining the Symbol Table
7-43 Setting Format Options
7-43 Debugging PostScript
7-44 Debugging Objective-C
7-48 Debugging Mach Threads
7-48 Debugging NeXT Core Files
7-49 Altering Execution
7-50 Defining and Executing Sequences of Commands
7-52 Legal Considerations

8-1 Chapter 8: Mach Object Files
8-4 The Mach Header
8-5 The Load Commands
8-12 Relocation Information
8-13 The Makeup of Executable Object Files

Index

Introduction

3 How This Manual is Organized

4 Conventions

4 Syntax Notation

5 Special Characters

6 Notes and Warnings

Introduction

This manual describes the essential tools for developing a NeXT™ application—these tools include the 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 *NeXT Developer's Library*; the illustration facing the first page of this manual shows the complete set of manuals in this Library.

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 (see "Suggested Reading" in the *NeXT Technical Summaries* manual).

This manual assumes you're familiar with the standard NeXT user interface. Some experience using a NeXT application would be helpful.

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

How This Manual is Organized

This manual contains the following eight chapters:

- Chapter 1, "Putting Together a NeXT Application," provides an overview of the procedures and tools 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 and in other manuals in the *NeXT Developer's Library.*

- Chapter 2, "The VT100™ Terminal Emulator: Terminal," describes NeXT's Terminal application, which you use to interact with a UNIX® shell from the NeXT workspace.

- Chapter 3, "The NeXT Mouse-Based Editor: Edit," describes NeXT's mouse-based text editor, with which you can create and edit ASCII or RTF text files.

- Chapter 4, "Developer Applications and Utilities," describes miscellaneous applications and utilities that are useful in programming on a NeXT computer. Using these applications, you can look into a running application and examine its data, measure the dynamic memory usage of an application, and get information about the processes running on your NeXT computer. This chapter also describes a shell-based interface to the PostScript® Window Server.

- Chapter 5, "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 6, "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 header files, macros, and conditionals. It also lists the options that can be used with the **cpp** (C preprocessor) command.
- Chapter 7, "The GNU Source-Level Debugger," describes how to debug a C program using GDB, the GNU debugger.
- Chapter 8, "Mach Object Files," describes the format of Mach object (also known as Mach-O) files. This format is used on NeXT computers instead of the standard UNIX 4.3BSD **a.out** format.

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.

Chapter 1
Putting Together a NeXT Application

1-3 The Application Development Process
1-4 Application Developer Tasks
1-5 Interface Builder Tasks

1-6 Components of an Application Project
1-6 The Project Directory
1-6 Source Code Files
1-7 The Main File
1-7 Class Definition Files
1-8 C and Objective-C Source Code Files
1-8 Other Source Files
1-9 Interface Files
1-9 The Icon Header File
1-10 Image Files
1-10 Sound Files
1-11 Project Management Files
1-11 The Project File
1-11 Libraries
1-11 NeXT Libraries
1-12 Custom Libraries

1-12 Putting the Components Together
1-12 The **make** Program
1-12 The Makefile
1-13 Interface Builder Makefiles
1-13 The Project Makefile
1-14 The Shared Makefile
1-16 The Preamble File
1-17 The Postamble File
1-18 Creating a Custom Makefile
1-18 Running the Makefile

1-18 Debugging
1-19 The Compiler and Link Editor
1-19 Compiler and Link Editor Warnings
1-20 Debugging Aids in the Executable File
1-20 Compiler and Link Editor Examples
1-23 Debugging the Executable File
1-23 Stepping Through the Program
1-25 GDB Tracing Techniques
1-26 Other Debugging Tools

1-27 Installing an Application
1-27 The Application Search Path
1-28 The Application Directory
1-28 File Packages
1-29 Choosing Extensions for an Application's Documents

Chapter 1
Putting Together a NeXT Application

NeXT computer applications, like other computer applications, are complex mechanisms. Before you put together such a mechanism, you need to carefully craft the pieces to be sure that each does its job correctly and that all fit together properly. Once this work is complete, you're ready to do the final assembly. This chapter introduces the tools that you use to assemble your carefully crafted pieces into a working application. Most of the tools described here are discussed in greater detail in the subsequent chapters of this volume; Interface Builder™ is discussed in the *Concepts* volume of the *Next Developer's Library.*

This chapter is organized into five sections:

- "The Application Development Process" describes the steps you take in creating a working application.
- "Components of an Application Project" describes the types of files that go into your application, where they come from, and how they relate to one another.
- "Putting the Components Together" covers the **make** program and the makefile, which provides **make** with the information it needs to create your application.
- "Debugging" summarizes techniques for debugging a NeXT application using the compiler, debugger, and other tools.
- "Installing an Application" discusses how to put your working application in a directory and make it available to the Workspace Manager™.

The Application Development Process

The process of developing an application can be divided into three main 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 using the NeXT Interface Builder, you can move easily between these tasks. Interface Builder lets you create and test "live" interface objects early in the design process. It creates skeletal class definition files for you to flesh out with working code, and parses your existing class definition code to make action methods and **id** instance variables available to interface objects. It also automates certain project management chores to make debugging easier.

The following discussion breaks the application development process down into sub-tasks, and categorizes them by responsibility: what you do, and what Interface Builder does for you.

Application Developer Tasks

When you create a NeXT application with the help of Interface Builder, you do the following:

- **Design your application.** Some design issues to consider are program structure, functionality, and user interface. You need to determine the unique classes that your application will require and think about how to divide your program into separate modules. You can use Interface Builder to design, prototype, and test user interface features.

- **Create a project directory for your development files.** You use the project directory to store the files associated with your application. These include source code files, interface files, sound and image files, and project management files. Interface Builder creates several files of its own that are stored in the project directory. You can create the directory from the Workspace Manager, from Interface Builder, or from a command shell such as the New Shell available from Workspace Manager's Tools command or the Terminal application described in Chapter 2 of this manual. The contents of the project directory are described in detail in the next section, "Components of an Application Project."

- **Write code for the unique classes of your application.** To establish the unique workings of your application, you create class definition files that include code for the appropriate methods, functions, and instance variables. Interface Builder can help you by creating skeletal code for a class if you list the methods in the Inspector panel; it can also parse the class definitions you write to let you use their methods in connections to other objects in your application.

- **Connect the objects in your application using Interface Builder.** You can drag links between objects in Interface Builder to create outlet connections; for example, to establish the target and action for a control in the interface.

- **Create icons and add them to your application.** Unless you want to keep the generic application icons provided by Interface Builder, you'll need icons to represent the application and its documents. You may want to add other unique icons for the buttons in your application using Interface Builder. You can also put images in custom views using Application Kit classes and methods and PostScript code.

- **Add sounds to your application.** You can add sounds to the buttons in the interface from within Interface Builder. You can add other sounds to the interface using Sound Kit™ methods.

- **Compile your program with the make program and the project makefile.** When you run the **make** program, it issues system commands to compile and link your application's source files into an executable file. The project makefile, generated by Interface 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.

- **Continue debugging your program using the GDB debugger and other debugging aids.** Once your application compiles successfully, you may want to run it from a shell using GDB, described in Chapter 7. GDB lets you control your application as it runs, examine and change data values, set breakpoints, and step through your code.

- **Install your program in an application directory.** When it's debugged, install the executable file in the application directory using one of the makefile options described later in this chapter or by copying the executable file to the directory. Once you do, it can be used just like any other NeXT application. If your application is intended for wider distribution, you may want to configure it using tools provided by NeXT so that a user can install it from floppy disks using the **Installer** program.

Interface Builder Tasks

As you work on your project, Interface Builder performs several tasks for you automatically:

- **Manages your application project.** Once you create a project directory and save your application in it, Interface Builder manages the files used by the project. The files it tracks include the main file, class definition files, icon files, sound files, and others. To keep track of these files, Interface Builder creates the file **IB.proj** in your project directory.

- **Archives your interface objects and their connections.** Interface Builder puts information on the classes used by your application—both Application Kit classes and unique classes you define—in the interface file. This file includes all the information required to generate the objects in your application at run time: specifications for Application Kit objects, connections between objects, icons, sounds, and other features. Interface Builder creates at least one interface file for each application you create.

- **Manages icons and other files that will be incorporated in the executable file.** If you add unique icons for your application and its document files, Interface Builder creates and maintains a custom icon header file that establishes the connection between your application and its icons. Interface Builder can also know about and manage other project files, such as standard C source code or sound files.

- **Maintains the project makefile used to put your application together.** As you add source files to your application, Interface Builder lists them in the project makefile. The **make** program reads the project makefile and generates the executable file from the sources.

Components of an Application Project

This section provides a closer look at the files that make up an application project, including source code and project management files. These files are stored in the project directory.

The Project Directory

The project directory provides you and Interface Builder with a convenient way to organize the files used in putting together your application. The project directory also provides a single path for all the files the **make** program uses to build your application. The **make** program is described in the next section.

The files in your project directory can be divided into three main categories: source code, other sources, and project management. The following table lists the various types of files in your project directory by category. Most files are identified by their extensions; the others are identified by a complete file name.

Category	File Type	Identified By
Source code	Main	*ApplicationName*_main.m
	Class Definitions	.h and .m
	Objective-C language code	.m, .pswm
	Other C source code	.c, .psw
Additional sources	Interface	.nib
	Icon header	*ApplicationName*.iconheader
	Images	.tiff
	Sound	.snd
Project management	Project	IB.proj
	Makefile	Makefile

Source Code Files

One source code file, the main file, is required for any application created with Interface Builder. To implement any unique features, your application also requires class definition files for its unique classes and methods. You may occasionally want to add source code files containing C source code for a particular purpose in an application.

The Main File

Each time you start a new application project with Interface Builder, it creates a file named *ApplicationName*_**main.m**. This file contains only the **main**() function required by every application written in C. A typical Interface Builder main file has the following contents:

```
#import <stdlib.h>
#import <appkit/Application.h>

void main()
{
    NXApp = [Application new];
    [NXApp loadNibSection:"MyApp.nib" owner:NXApp];
    [NXApp run];
    [NXApp free];
    exit(0);
}
```

The **main**() function performs several tasks. It creates an instance of the Application class and assigns its **id** to the global variable NXApp, so that all objects in the application can access the Application object. It loads the objects specified in the interface section of the application's executable file. The function then begins the main event loop by sending the application the **run** message. When the Application object receives a **terminate** message (for example, from the Quit MenuCell in the main menu), **main**() cleans up the run-time environment and halts.

Class Definition Files

Class definition files provide the methods, functions, and variables for the unique classes in your application. Each class is defined with a pair of files. The class interface file, with the extension ".h," describes the class's interface to other source modules. The implementation file, with the extension ".m," contains the actual code that implements the class and instance methods.

Interface Builder lets you add classes to your application in one of two ways. You can write the code for your application's classes first, then parse the files so that Interface Builder knows about their methods. Alternatively, you can define classes and methods using the Class Inspector, then unparse them to create templates in which you later write code.

There are several guidelines to keep in mind as you're defining classes for your application:

- For greater modularity, write your program's unique code in subclasses of Object or View. If you create a subclass of Application to contain your unique code, your program loses modularity. Only one Application instance is allowed in each program, so you can't easily move code from one program to another when it's in a subclass of Application.

- Define the interface and methods for each new class in one pair of files. For example, if you create a subclass of Object called Translator, create the files **Translator.h** and **Translator.m** to define the interface and methods, respectively. While you can define several classes in one pair of files, you should generally do so only if one class is public and the others are private to that class. If you define several public classes in one source file, you inhibit modularity.

- You can also use several files to define one class. This lets you make both objects and methods modular; you might, for example, group methods for printing in one pair of source files and group methods for file management in another pair of source files.

- When you name classes, begin the name with an uppercase letter. For example, if you create a subclass of View, name it "AnotherView" rather than "anotherView."

- When you name instances, begin the name with a lowercase letter. For example, if you create an instance of "AnotherView," name it "myOtherView" rather than "MyOtherView."

C and Objective-C Source Code Files

If your application uses source code files that aren't class definitions, you can add them to your project with Interface Builder by listing them in the Project Inspector Files display. Files containing only standard C code are added as ".c" files; files containing Objective-C language code are added as ".m" files. If you use C or Objective-C language source files that contain PostScript code to be processed by the pswrap program, add them to the ".psw" or ".pswm" files list in the Files display. When the application is compiled using the project makefile created by Interface Builder, these other source files are also compiled.

Other Source Files

NeXT applications frequently use information in addition to source code, such as image and sound files. The Mach object-format executable file that the link editor creates from your source files can include this additional information. These additions in the executable file are organized into *segments*; the individual entries within segments are stored in *sections*. When you work with Interface Builder, it creates two files that are written into sections: the interface file and the icon header file. It also manages image and sound files that are copied into segments.

Interface Files

As mentioned earlier, Interface Builder maintains interface files (with the extension ".nib") to store information on the Application Kit classes in your application and the unique subclasses of Object or View that you define. Interface Builder creates an interface file for each application and for each new module you add to an application. Data from the interface files is copied into the __NIB segment of the file when the program is linked.

The Icon Header File

The icon header file (identified by the extension ".iconheader") contains information about application and document file icons. If you identify custom icons in the Project Inspector, Interface Builder creates a custom icon header file; otherwise it lists the generic application icon in the icon header file.

Information in the icon header file is copied into the header section of the __ICON segment of the executable file when the application is linked; the icon bitmaps are copied into the other sections of this segment. The Workspace Manager uses the icon header to manage the display of icons in the File Viewer, the dock, and elsewhere in the NeXT user interface.

An icon header file contains two types of lines: F lines and S lines. The following example shows a typical icon header file with two lines:

```
F    Write        Write        app
S    text         Write        text
```

The entries on the F line are as follows:

- The first entry, "F," identifies this as a line of information about the icon for the application.
- The second entry, "Write," is the name of the application's executable file. Since no path is listed, the Workspace Manager will look for the application in the default search path. The search path is discussed at the end of this chapter.
- The third entry, "Write," is the name of the application. This entry differs from the second entry if the name of the application is different from that of its executable file.
- The fourth entry, "app," is the name of the section in the __ICON segment where the application icon bitmap is stored. This name is always "app;" the Workspace Manager requires this name.

The entries on the S line are as follows:

- The first entry, "S," identifies this as a line of information about document file icons. "S" stands for suffixes, since an application's document files are identified by suffixes (extensions).

- The second entry, "text," identifies the extension used for document files.

- The third entry, "Write," is the name of the application to launch when the file is opened by double-clicking its icon.

- The fourth entry, "text," is the name of the section in the __ICON segment where the document icon bitmap is stored. Interface Builder uses the name of the file containing the bitmap for the icon, minus any extension it may have.

When Interface Builder creates the icon header file for an application, it can contain several lines, including one F line for the application icon and up to three S lines for document file icons. The Project Inspector Attributes display provides spaces for identifying these three icons.

By creating your own icon header file, you can specify additional icon information. For example, you would create your own icon header file if your application had more than three document file types.

Image Files

Custom images can make your user interface easier to use and distinguish your program from other NeXT applications. You can create an icon or image with any graphics program that creates TIFF or EPS files. To add icons for your application or document files, use the Attributes display of the Interface Builder Project Inspector.

You can put other image sources for your application in Interface Builder's Project Inspector. If you write custom code that uses TIFF image files, add them to the ".tiff" files in the Files display of the Interface Builder Project Inspector; they are placed in the __TIFF segment of the executable file.

Sound Files

To add a sound to a button, drag the sound file into Interface Builder's Sounds window. To include sounds in other parts of your program, you need to write custom code using Sound Kit methods; you can then add the files used by these methods to the ".snd" files in the Project Inspector's Files display.

Project Management Files

Interface Builder maintains two project management files automatically: the project file and the makefile. The project file is described here; the makefile is described with the **make** program in the section "Putting the Components Together."

The Project File

Interface Builder creates and maintains the project file to track the elements of an application. The project file, always called **IB.proj**, records all the source files in the application. These include the main file, class definition files, interface files, icon files, and sound files described previously; they may also include other source files, if you list them in the Project Inspector's Files display. Interface Builder uses information in the project file to create the icon header file and makefile for the project.

Libraries

Libraries contain compiled code for previously defined classes, methods, and functions that your application is based on. This section briefly discusses the libraries you may need to access from your application.

NeXT Libraries

The NeXT libraries are contained in the directory **/usr/lib** and **/lib**. These libraries contain all predefined classes, methods, and functions for the NeXT Application Kit, Sound Kit, and Music Kit™, as well as the C and Objective-C language libraries on which they are based. There are two types of libraries: standard and shared.

When you use standard libraries in your application, the compiled library code is placed in the executable file wherever a library method or function is used. When you use shared libraries, the code remains in the library and is mapped into your application's address space at run time; the libraries are thus shared among running applications. Applications created with Interface Builder automatically include shared libraries.

If you use Music Kit classes in your application, you must explicitly include the libraries for these classes and methods. The technique for doing this is described in the section "The Preamble File."

Custom Libraries

You can create and compile your own library files for use with your applications; you can also include methods and functions from other custom libraries. If you reference other libraries in your application, include them by creating a preamble file as described in the next section.

Putting the Components Together

Once you've created the components of your application and collected them in the project directory, you're ready to build the working application. You could use system commands to compile each of your source files, then link the resulting object files to make your application. However, the **make** program automates this process. **make** creates your application using information in the makefile, which describes how to compile and link all the elements of your application into an executable file.

The make Program

make is a standard UNIX program for managing *sources* and generating *targets*. The sources are all the files in your project directory that will be compiled and linked into your executable file, including the main file, the class definition files, image and sound files, and the interface file. The targets that **make** generates from these sources include your executable file and any intermediate files, such as object files, created in the process of generating your executable file. **make** works by issuing system commands that generate the targets from the sources.

In managing targets and sources, **make** keeps track of source updates. Each time you run **make**, only the targets whose sources 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.

The Makefile

The makefile lets **make** know about the targets and sources in a project. The makefile includes two types of information useful for managing a project: *dependencies* and *macros*.

Dependencies define targets, their sources, and the system commands needed to generate the targets from the sources. A simple dependency is:

```
MyApp.o: MyApp.c
    cc -c MyApp.c
```

In this example, **MyApp.o** is the target, **MyApp.c** is its source, and the system command **cc** runs the compiler and generates the object file from the C source file.

Macros define substitutes for targets, sources, command-line options, and other information required to create a target from its sources. A typical macro is:

```
CLASSES = MyClass.m AnotherClass.m
```

In this example, the macro CLASSES is defined to substitute for the name of two class implementation files, **MyClass.m** and **AnotherClass.m**.

Interface Builder Makefiles

Interface Builder uses a set of four makefiles with the **make** program. For each application, Interface Builder creates a project makefile, named **Makefile**, in the project directory. All Interface Builder applications use the shared makefile, **app.make**, stored in the directory **/usr/lib/nib**. You can customize the way your program is compiled and linked by creating two auxiliary files, **Makefile.preamble** and **Makefile.postamble**, in the project directory.

The Project Makefile

When you save an application, Interface Builder generates a project makefile, with the name **Makefile**, in the project directory. The project makefile defines specific macros to represent the custom objects in your project. For example, the macro NAME is always defined with the name of your application's executable file. If you include additional files in the Project Inspector Files display, such as sound files or C source files, Interface Builder adds macro definitions for these files to the project makefile.

The purpose of the project makefile is to provide application- and environment-specific macro definitions to **app.make**. **app.make** does the actual job of compiling and linking your application.

The project makefile follows a standard format. In the example below, lines listed in bold are customized by Interface Builder for your application; those in regular type are standard for all project makefiles generated by Interface Builder.

```
#
# Generated by the NeXT Interface Builder.
#
# NOTE Do NOT change this file -- Interface Builder maintains it.
#
# Put all of your customizations in files called Makefile.preamble
# and Makefile.postamble (both optional), and Makefile will
# include them.
#
```

```
NAME = MyApp

INTERFACES = MyApp.nib
CLASSES = MyClass.m
MFILES = MyApp_main.m
APPICON = MyApp.tiff
DOCICONS = Xtra.tiff

SOURCEMODE = 444

LIBS = -lNeXT_s -lsys_s
DEBUG_LIBS = $(LIBS)
PROF_LIBS = -lNeXT_p -lsys_p

MAKEFILEDIR = /usr/lib/nib/
ICONSECTIONS = -sectcreate __ICON app MyApp.tiff

INSTALLDIR = $(HOME)/Apps
INSTALLCFLAGS = -c -s -m 755

-include Makefile.preamble

include $(MAKEFILEDIR)/app.make

-include Makefile.postamble

-include Makefile.dependencies
```

app.make is included in every project makefile with the line:

```
include $(MAKEFILEDIR)app.make
```

The project makefile includes **Makefile.preamble** just before **app.make** and **Makefile.postamble** just after. Both files are *conditionally* included using the **-include** option; they're included only if present in the project directory. If you have created a dependencies file using the **depend** target described in the next section, it is also included by the project makefile.

The Shared Makefile

app.make is the shared makefile used to generate the executable file for all applications created with Interface Builder. This file resides in **/usr/lib/nib**. The dependencies and macros in **app.make** are based on the macros defined in the project makefile. **app.make** 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, give the command

```
make target
```

from the shell command line.

The following table lists each target and its task.

Target	Task
[*ApplicationName*]	Compiles and links an optimized version of the project, ready to install. *ApplicationName* is the name listed in the Project Inspector Attributes display. This is the default target used when you give the **make** command from the command line without a target. You can choose this target for Interface Builder's Make command by clicking the Make button in the Project Inspector Panel.
debug	Generates the executable file *ApplicationName***.debug**; the extension .debug indicates that the file contains symbols for debugging the application with the GNU debugger. This target also sets the compiler flag **-DDEBUG** to generate conditionally compiled debugging code. The **debug** target is the default target used by the Make command from Interface Builder's File menu. More information on debugging can be found later in this chapter and in Chapter 7, "The GNU Source-Level Debugger."
clean	Removes all derived files, such as object and executable files, from the project directory, returning the project to its precompiled state.
install	Moves the application into the installation directory specified in Interface Builder; the default is **$(HOME)/Apps**.
installsrc	Installs the source files for the project into another directory. You must specify the target directory on the command line after this target. If it exists, the directory and its contents will be deleted; the directory will 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. Once this file exists in the project directory, it's conditionally included by your project makefile.
diff	Compares the source of the current project with another project and displays the differences in the shell window.
profile	Generates 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.

These targets work by calling **make** recursively. That is, they define the dependencies for performing their assigned task, then start the **make** program again using those dependencies.

In addition to the macros defined in the project makefile, **app.make** accepts macro definitions from **Makefile.preamble**. Most of the macros that you can define in **Makefile.preamble** begin with "OTHER." These macros cause **make** to perform operations and work on files other than those specified in Interface Builder's project makefile. The macros are:

Macro Name	File Type
CFLAGS	Compiler flags not set by **app.make**. A complete list of compiler flags can be found in Chapter 5.
LDFLAGS	Link editor flags not set by **app.make**. A complete list of link editor flags can be found in Chapter 5.
OTHER_SOURCEFILES	Source code files not listed in the Files display in Interface Builder's Project Inspector.
OTHER_GARBAGE	Derived files not generated by the dependencies in **app.make**; files in OTHER_GARBAGE are deleted with the **clean** target.
OTHER_OFILES	Other object files referenced in your application code.
OTHER_LIBS	Other libraries referenced in your application code; an example of using OTHER_LIBS is included below.
OTHER_PROF_LIBS	Other profile libraries; for use with the **profile** target.

Before you modify the Interface Builder's makefiles with **Makefile.preamble** or **Makefile.postamble**, it may help to look at **app.make**. Open it with the Edit application, then examine its dependencies to see how they use the macros listed above. Exploring **app.make** may answer questions about how to expand the scope of **make** using **Makefile.preamble** and **Makefile.postamble**.

The Preamble File

Makefile.preamble lets you modify the macros from **app.make** in a couple of ways: by overriding macros defined in the project makefile and by defining the macros listed above.

To override a macro definition from 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
```

Of course, you need write permission for the directory to install an executable in it.

To use one of the macros listed above in **app.make**, you first define it in **Makefile.preamble**. You would do this, for example, when creating a Music Kit application with Interface Builder. The shared libraries used by Interface Builder applications don't include the Music Kit libraries, so you would define the macro OTHER_LIBS in **Makefile.preamble**:

```
OTHER_LIBS = -lmusic_s -lmidi -ldsp_s
```

The CFLAGS and LDFLAGS macros lets you set the compiler and link editor flags in **Makefile.preamble**. For example, if you write an application that uses the Music Kit's Pluck instrument, add the following link editor flag to your **Makefile.preamble**:

```
LDFLAGS = -u .objc_class_name_Pluck
```

You can also define link editor flags to add segments to your executable file. For example, the sample application Draw defines the following macro in its **Makefile.preamble**:

```
LDFLAGS = -segcreate HELP document help.draw
```

Using this macro definition, the link editor will create a segment named "HELP" in the executable file; that segment will have a section named "document" containing the document file **help.draw**. For more on compiler and link editor flags, see the next section of this chapter and Chapter 5.

The Postamble File

To alter or add to the dependencies used by **make** as it creates your program, create a file called **Makefile.postamble** in the project directory. This file is included by the project makefile after **Makefile.preamble** and **app.make**.

One use for **Makefile.postamble** is to define dependencies that generate OTHER_OFILES from OTHER_SOURCEFILES. **app.make** doesn't define such a dependency, since OTHER_SOURCEFILES can be any source file type. If, for example, the OTHER_SOURCEFILES were assembly language source code files for the DSP, you would add a dependency in **Makefile.postamble** to generate OTHER_OFILES by running the assembler.

Creating a Custom Makefile

If you're not using Interface Builder for creating and managing your application, you'll find it convenient to create your own makefile to generate the executable file from your source files. For examples of makefiles, look through the project makefiles generated for the example applications by Interface Builder, and look at **app.make** in **/usr/lib/nib**.

Running the Makefile

When your makefiles are in order, you're ready to run the **make** program. You can run **make** in two ways: from Interface Builder or from a shell.

When you choose the Make command from the File menu, Interface Builder sends commands to the system to start up a shell, and then runs **make** in the shell using the target selected in the Project Inspector panel (debug is the default target).

You can also run the **make** program directly in a shell. To run **make**, change your working directory to the project directory, then type the **make** command. If you've created your project with Interface Builder, you can use any of the **make** targets described above. Until your application is running correctly, you can use **make debug** for the debugging aids it provides; these are described in the next section.

When you start **make**, it begins generating targets from sources using the compiler and link editor. As they run, the compiler and linker generate messages and warnings about your source files. At this point you begin debugging your application.

Debugging

This section summarizes debugging tools available on NeXT computers and how to begin debugging your application.

If you're an experienced programmer, you've developed your own debugging strategies. Most of these apply to debugging on NeXT computers. The material here is intended to help you expand your strategies with new techniques for debugging Objective-C language programs. For this reason, this section focuses on general things to look for and do when debugging NeXT applications. There are also a few pointers about specific problems you may encounter in debugging.

This section only introduces the compiler, link editor, and debugger. For complete references on these programs, see Chapters 5 and 7.

The Compiler and Link Editor

The compiler and link editor aid the debugging process in a couple of ways:

- They generate warnings as they run to tell you about errors and anomalies in the source code.
- They can compile your executable file in ways that make it easier to trace errors in the source code.

Compiler and Link Editor Warnings

Once you start **make**, pay attention to the messages that the compiler and link editor display in the shell as they run. Compiler messages and warnings are useful for pinpointing problems in your code. In general, there are three levels of compiler warnings:

- Nonfatal anomalies will produce warnings without halting the compiling process. The warnings for such anomalies generally include the word "may," as in "You may be using an uninitialized variable."
- Serious but nonfatal code problems may generate warnings but allow compiling to continue with certain assumptions. This happens if, for example, your source code doesn't import the header for a class that it sends messages to. The compiler proceeds, making assumptions about the argument and return types for the method being used.
- Fatal errors produce warnings and stop the compiler before it generates your executable file. Fatal errors include compiler errors such as illegal syntax, and linking errors such as the inability to find a library for a particular class, method, or function.

Of course, you must fix fatal errors. Beyond that, you should understand nonfatal errors and the warnings they generate. Be sure your code is doing what you want it to do. For example, you can leave unknown methods in your code if you choose: Dynamic binding defers selecting the object to receive a message until run time. The compiler warns that it doesn't know the method, but continues creating your executable file. This shifts some of the burden for type checking to you. If you don't use strict typing, be sure your methods and arguments match the objects that will receive them. If your program generates a mismatch, you'll get a run-time error.

The best way to deal with warnings is to fix the cause, even if the problem isn't severe. If you don't fix the code, at least add a comment to describe the warning it generates. This approach is useful for a couple of reasons. For one, you'll have a hard time telling new warnings from old if you modify the program. For another, if you aren't the only programmer working on the project, the next person encountering those warnings will have a hard time knowing where to start to fix real problems.

All of the targets in Interface Builder's **app.make** file set the compiler warning flag **-Wimplicit**. For more complete warnings, add the following macro definition to your **Makefile.preamble**:

```
CFLAGS = -Wall
```

This defines the compiler flag macro CFLAG as **-Wall**, causing the compiler to display all warning messages.

The compiler and link editor flags are discussed in Chapter 5.

Debugging Aids in the Executable File

When you run **make** using debug as the target, the dependency defines two compiler flags that place debugging aids in your executable file.

The **-g** flag puts symbols in your executable file that are used by the GNU debugger, GDB. When you run GDB, it uses this information to let you examine the source code as your program executes.

The **-DDEBUG** flag allows you to conditionally compile lines of your source code for debugging. With the **-DDEBUG** flag set, the compiler places the debugging code in your executable file; without the **-DDEBUG** flag, the compiler leaves the debugging code out.

To place conditionally compiled debugging code in your source file, put it between the compiler directives **#ifdef DEBUG** and **#endif**. For example:

```
#ifdef DEBUG
    printf("**Debug: myMethod: returns %d\n",[self myMethod:]);
#endif
```

This conditionally compiled code causes the object defined by the source code to display the return value of the method **myMethod** in the shell.

Compiler and Link Editor Examples

This section offers four examples of compiler and link editor runs. With the accompanying descriptions, these examples should help you understand compiler and link editor messages and how to respond to them.

The first example demonstrates a successful compilation, with commands echoed by the **make** program as it processes the makefiles:

```
> make debug
make Hello.debug "OFILE_DIR = debug_obj"  "CFLAGS =  -g -DDEBUG
-Wimplicit"
cc -g -DDEBUG -Wimplicit  -c Greeter.m -o debug_obj/Greeter.o
cc -g -DDEBUG -Wimplicit -segcreate __ICON __header Hello.iconheader
-segcreate __ICON app Hello.tiff -segcreate __NIB Hello.nib Hello.nib
-o Hello.debug debug_obj/Greeter.o debug_obj/Hello_main.o -lNeXT_s
-lsys_s
```

In this example, the first command generated by the **make** program with **debug** as the target is:

```
make Hello.debug "OFILE_DIR = debug_obj"  "CFLAGS =  -g -DDEBUG
-Wimplicit"
```

This command recursively calls the **make** program to make the target file **Hello.debug**. Object files will be in a subdirectory **debug_obj**. The compiler flags are set to **-g**, **-DDEBUG**, and **-Wimplicit**.

The second command runs the compiler, generating the object files from the source files:

```
cc -g -DDEBUG -Wimplicit  -c Greeter.m -o debug_obj/Greeter.o
```

The third command runs the link editor, which creates the executable file from the object files, interface files, TIFF files, and other components.

```
cc -g -DDEBUG -Wimplicit -segcreate __ICON __header Hello.iconheader
-segcreate __ICON app Hello.tiff -segcreate __NIB Hello.nib Hello.nib
-o Hello.debug debug_obj/Greeter.o debug_obj/Hello_main.o -lNeXT_s
-lsys_s
```

Once these three commands are processed, the executable is complete and **make** returns.

The next example shows a nonfatal compiler warning. Differences from the above example are in bold.

```
make Hello.debug "OFILE_DIR = debug_obj"  "CFLAGS =  -g -DDEBUG
-Wimplicit"
cc -g -DDEBUG -Wimplicit  -c Greeter.m -o debug_obj/Greeter.o
Greeter.m: In method 'setOutputForm:'
Greeter.m:12: warning: cannot find method.
Greeter.m:12: warning: return type for 'setOutputForm:' defaults to id
cc -g -DDEBUG -Wimplicit -segcreate __ICON __header Hello.iconheader
-segcreate __ICON app Hello.tiff -segcreate __NIB Hello.nib Hello.nib
-o Hello.debug debug_obj/Greeter.o debug_obj/Hello_main.o -lNeXT_s
-lsys_s
```

Here, the compiler detects a message being sent to an unknown class: It can't find the method **setOutputForm:** in the header files it has access to. However, this doesn't prevent the program from compiling. The compiler lets the return type of the method default to **id**, an object identifier type. When you get this type of warning, check that your source code imports the correct class header files and that you've used the correct method name.

In the next example, the compiler detects a fatal error.

```
make Hello.debug "OFILE_DIR = debug_obj"  "CFLAGS =  -g -DDEBUG
-Wimplicit"
cc -g -DDEBUG -Wimplicit  -c Greeter.m -o debug_obj/Greeter.o
Greeter.m: In method 'setOutputForm:'
Greeter.m:14: incompatible types in argument passing
*** Exit 1
Stop.
*** Exit 1
Stop.
```

Here, the header file missing in the previous example has been included. This allows the compiler to perform type checking which results in the detection of an incompatible argument type.

Finally, the following example is based on an Interface Builder application that includes Music Kit classes. It demonstrates a fatal error from the link editor.

```
> make debug
make HelloNote.debug "OFILE_DIR = debug_obj"  "CFLAGS =  -g
-DDEBUG -Wimplicit"
mkdirs debug_obj
cc -g -DDEBUG -Wimplicit  -c ExampApp.m -o debug_obj/ExampApp.o
cc -g -DDEBUG -Wimplicit  -c HelloNote_main.m -o
debug_obj/HelloNote_main.o
cc -g -DDEBUG -Wimplicit -segcreate __ICON __header
HelloNote.iconheader -segcreate __ICON app
/usr/lib/nib/default_app_icon.tiff -segcreate __NIB HelloNote.nib
HelloNote.nib -o HelloNote.debug debug_obj/ExampApp.o
debug_obj/HelloNote_main.o -lNeXT_s -lsys_s
/bin/ld: Undefined symbols:
_MKNoteTag
_MKKeyNumToFreq
.objc_class_name_Note
.objc_class_name_Orchestra
.objc_class_name_SynthInstrument
.objc_class_name_Pluck
.objc_class_name_Conductor
*** Exit 1
Stop.
*** Exit 1
Stop.
```

Here, the warnings in bold indicate that there are missing symbols: variables and class names. These are defined in the Music Kit libraries. As mentioned in the last section, these libraries must be included with a line in **Makefile.preamble** when you create a Music Kit application in Interface Builder. Similar errors are generated any time the link editor doesn't have access to a library whose classes you reference.

Debugging the Executable File

After you successfully compile your program, you're ready to try running it. You can do this in a couple of ways:

- GDB, the GNU source-level debugger, offers an interactive environment for controlling and testing your program. The version of GDB provided with NeXT computers has been modified specifically for debugging Objective-C language methods and objects. Chapter 7 describes the variety of controls, operating modes, and options available in GDB. This section highlights some of the program's more frequently used features.
- Rather than running in GDB, you may choose simply to run your program until it fails, then check the source code for errors at the point where it failed. If you do this, it's useful to include conditionally compiled debugging code in the executable file, as described earlier in this section.

In either case, you may want to run your program from a shell. The shell lets you scroll through, print, copy, and paste the messages that your application generates at run time.

Stepping Through the Program

When you run your program in GDB, you can step through the program looking for problems. When you encounter an error, you may want to rerun the program and examine the error more closely. This section describes stepping through your program; the next section describes GDB tools for tracing errors.

Note: Most GDB commands have one- or two-letter abbreviations that serve as substitutes. These are included in parentheses immediately following the command name.

A typical session in the debugger might follow these steps:

1. Start the shell, change your working directory to the project directory, then type:

```
> gdb ApplicationName
```

2. Give the **run** (**r**) command at the "gdb" prompt. The program starts up and runs until it encounters an error, then halts. GDB displays an error message describing the type of error and the name of the method or function where the error occurred.

3. Use the **list** (**l**) command to list the method or function. You can list code by method name, function name, or line number. GDB responds with the source code and line numbers. If you don't include a line number or method name after **list**, GDB lists the most recently executed line of code in your source files. The list includes ten lines surrounding the target line:

```
(gdb) l setOutputForm:
5       #import <appkit/Form.h>
6
7       @implementation Greeter
8
9       - setOutputForm:anObject
10      {
11          outputForm = anObject;
12          Greeting = "Hello, World";
13          [outputForm setStringValue:Greeting at:0];
14          return self;
```

4. Give **breakpoint** (**b**) commands to set breakpoints around an error. You can set a breakpoint by method name, function name, or line number. If you use line numbers, GDB assumes you're referring to the last file listed or executed; to override this, give the file name and line number, separated by a colon:

```
(gdb) b Greeter.m:20
```

If you use method names and GDB detects several source files with the same method, it lets you select the appropriate one:

```
(gdb) b setStringValue:
The following classes implement the selector:
1.  -ActionCell
2.  -ButtonCell
3.  -Cell
4.  -Control
5.  -SliderCell
Enter the number of the class you want:
```

Once set, a breakpoint applies until you explicitly clear it or until you exit GDB.

5. Run the program again.

6. When execution reaches a breakpoint, the program halts. Give the **continue** (**cont**) command to proceed to the next breakpoint, or give the **step** (**s**) or **next** (**n**) command to execute one line at a time. **step** goes through every line of source code, including library methods and functions. **next** goes through code a line at a time, but skips over function calls without stopping.

7. When you encounter the error, use the tracing techniques described in the next section to focus on the problem. You may need to rerun the program and step through again, examining your program until you pinpoint the problem.

8. When you've solved one problem, run the program and repeat the process until you track down all the errors.

GDB Tracing Techniques

The following GDB commands can help you trace errors by letting you examine objects, methods, instance variables, and other elements of your program's run-time environment.

The **print** (**p**) command lets you see the values returned by a message or stored in an object's instance variables. To convert data types, explicitly cast the type of the return value. For example:

```
(gdb) print [outputForm stringValueAt:0]
$19 = 84208
(gdb) print (char *)[outputForm stringValueAt:0]
$20 = (char *) 0x148f0 "Hello, World"
```

By default, the return value of this method is cast as an integer; explicit casting converts it to a character string.

The **set** command (no abbreviation) lets you send messages to objects and alter the values stored in instance variables. For example:

```
(gdb) set [outputForm setStringValue:"Buenos Dias" at:0]
(gdb) print (char *)[outputForm stringValueAt:0]
$21 = (char *) 0x148f0 "Buenos Dias"
```

This **set** command replaces the previous value "Hello, World" with "Buenos Dias."

The **backtrace** (**bt**) command shows how your program got where it is by showing you the frame stack: the series of methods and functions that led to the current state. The outermost frame on the stack is your program's **main()** function; the innermost frame is the method or function where the program is currently executing.

Backtrace may be useful for tracing the flow of events and messages as they pass between objects in your application. On the other hand, many frames in the frame stack may be produced by library methods rather than methods in your own code. These appear on the frame stack without source code references. For example:

```
(gdb) bt
#0  - [Greeter=0x000127f0 greetInGerman: sender=(id) 0xe470]
(Greeter.m line 16)
#1  0x5008a22 in - [Object perform:with:]
#2  0x601aa16 in - [Control sendAction:to:]
#3  0x602b862 in - [Matrix sendAction:to:]
#4  0x602d8b4 in trackCell ()
#5  0x602db9a in trackMenu ()
#6  0x602e7a6 in - [Menu mouseDown:]
#7  0x602ee7e in - [MenuCell trackMouse:inRect:ofView:]
#8  0x602a7d0 in - [Matrix _mouseDown_normalMode:]
#9  0x602b474 in - [Matrix mouseDown:]
#10 0x6051df0 in - [Window sendEvent:]
#11 0x60086f2 in - [Application sendEvent:]
#12 0x600788e in - [Application run]
#13 0x276a in main (argc=1, argv=(char **) 0x3fffd40)
(Hello_main.m line 12)
End of backtrace: saved frame pointer is zero.
```

In this backtrace, the frames in bold represent custom source code; their listings include the method or function name, the source file name, and the line number. The other frames are generated by objects and methods from the Application Kit. To debug your application, you need to first locate your code on the frame stack. Then you can begin to determine if the correct message is being sent, if the correct object is receiving it, and if the code you've written allows that object to respond properly to that message.

To review the backtrace entry for a frame, give the **frame** (**f**) command. For a more complete description of a frame, including register listings and other internal details, give the **info frame** (**i f**) command.

This list of GDB features is by no means complete. Other GDB commands allow you to list source files and symbols (object, method, function, and variable names), define variables, examine PostScript code, execute commands at breakpoints, and much more. See Chapter 7 for a complete discussion of GDB features.

Other Debugging Tools

Along with the compiler and GDB, the NeXT development environment includes several applications and features that can help you trace your program and pinpoint errors.

The applications described in Chapter 4, AppInspector™, MallocDebug, ProcessMonitor, and **pft**, provide additional insights into the workings of your program. The AppInspector lets you view the object hierarchy, examine instance variables, and observe messages being received by objects. ProcessMonitor lets you examine various characteristics of any

process's activities: memory use, PostScript graphics states, the run-time environment, and so on. MallocDebug measures the dynamic memory use of an application. **pft** lets you send PostScript code directly to the Window Server.

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 the shell; on the command line, enter the program name followed by the parameter.

Installing an Application

When your application is thoroughly tested and debugged, you're ready to use it. To do so, put it in a directory where the Workspace Manager will be able to find it. This section summarizes several Workspace Manager features to consider when installing your application.

The note **Installer.rtf** in the directory **/NextLibrary/Documentation/NextDev/Notes** describes how to prepare large applications for distribution on floppy disks.

The Application Search Path

You can start an application on a NeXT computer in several ways. When you start an application by typing its name in the shell, or by opening a document file from the File Viewer, the Workspace Manager has to find the executable file for that application. It looks for the executable file in a systematic sequence of directory paths, beginning with the current directory. This search sequence is contained in an environmental variable **path**.

Because of this search sequence, 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**. You should consider the path when naming and installing applications.

The Application Directory

You can put your program's executable file in any directory. It must be in a directory in the Workspace Manager's search path in order for its icons appear in the File Viewer.

Use **make install** to put your application in the default application directory set in the Interface Builder's Project Inspector: **$(HOME)/Apps**. Change the entry in the Project Inspector's Attributes display if you want **make install** to put your application in another directory.

Use **make** with no target to create the optimized executable file in the project directory. You can leave the file in the project directory or copy it to any other directory you choose. To make it available to the Workspace Manager, copy the executable file to a directory in the search path.

File Packages

A file package is a special kind of directory containing a set of related files that usually aren't accessed individually. You can use a file package to hold your application's executable file, along with associated files such as help files. A file package provides a convenient way to shield the files related to the application from the user; the package acts as though it were the application. The Workspace Manager represents the package in the File Viewer by the application icon, and using the Workspace Manager Open command starts the application rather than opening the directory.

You can create a file package automatically from Interface Builder's Attributes Inspector. To create a file package by hand, create a directory with the name of the application and the extension ".app" in your application directory. Choose the Open as Folder command from the Workspace Manager's File menu (or type Command-Shift-O) to open the directory. Then copy the executable file and other files for the application into the directory.

To associate the icon for your application with the file package, you need to include another F line in the icon header file to refer to the directory, as illustrated here:

```
F    Write.app    Write    app
```

The entries on the F line for a file package are as follows:

- The first entry, "F," identifies this as a line of information about the icon for the executable file.
- The second entry, "Write.app," is the name of the file package containing the executable. Since no path is listed, the default search path is used.
- The third entry, "Write," is the name of the application.

- The fourth entry, “app,” is the name of the section where the application icon bitmap is stored. This name is always “app” for Interface Builder projects.

Choosing Extensions for an Application’s Documents

For your application’s documents to work with the Workspace Manager, you need to plan ahead and write file management code that saves the documents with a unique extension. 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.

Chapter 2
The VT100 Terminal Emulator: Terminal

2-4 Introduction to Terminal

2-4 Starting Terminal

2-4 Setting Preferences
2-6 Emulation Preferences
2-7 Window Preferences
2-8 Shell Preferences

2-9 The Main Menu
2-9 Info
2-9 Shell
2-9 Edit
2-10 Format
2-10 Windows
2-10 Print
2-11 Services
2-11 Hide
2-11 Quit

2-11 The Shell Menu
2-11 New
2-12 Steal Keys
2-12 Clear Buffer

2-12 The Edit Menu
2-12 Cut, Copy, and Paste
2-13 Find
2-13 Select All

2-13 The Format Menu
2-14 Font Panel
2-14 Page Layout

2-14 The Find Menu
2-14 Find Panel
2-15 Find Next, Find Previous, and Enter Selection
2-15 Jump to Selection

Chapter 2
The VT100 Terminal Emulator: Terminal

Although you can run standard UNIX programs and commands on a NeXT computer, such programs aren't designed to be run directly from the NeXT 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.

For running such programs and commands on a NeXT computer, two methods are provided by NeXT: the Terminal application and the Workspace Manager's shell window. These two methods provide many of the same features, but there are minor differences that may lead you to choose one or the other for a particular purpose.

A Terminal window and a Workspace Manager shell window both offer the following features:

- Scrollers let you scroll backward to text that has already disappeared from the window.
- Text can be copied and pasted. You can copy and paste within a particular window, between windows, or even to and from other applications that support cutting and pasting, such as Mail and Edit.

Terminal also offers some additional features not found in a Workspace Manager shell window:

- Terminal has a Print command to let you print the contents of the window, and a Find command to let you search for text.
- Terminal's standard Services menu lets you make inter-application requests, such as defining a word in Digital Webster™ or searching for references in the Digital Librarian™.
- Terminal's Preferences command allows you to change the default size, 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 the features of 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 program runs, it prompts you for commands, interprets commands that you type, and passes these commands to the operating system for execution. For more information about the two most common UNIX shell programs, the Bourne Shell and the C Shell, see their UNIX manual pages (**sh** and **csh**).

Terminal provides a simple yet effective way to interact with a UNIX shell from the NeXT workspace. Terminal gives you access to UNIX commands and programs that can't be run directly from the workspace, and also lets you integrate shell input and output with other NeXT applications running in the workspace.

Starting Terminal

You can 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 enter the workspace. When Terminal starts up it creates one new Terminal window using the default Preferences settings. You can create additional Terminal windows as you need them. To create a window with different settings, select the appropriate settings in the Preferences panel before choosing the New Shell command.

Setting Preferences

The Preferences command in the Info menu displays the Preferences panel, shown in Figure 2-1. The Preferences panel lets you set default values for various Terminal options. For example, you can set default font properties or specify the size of 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.

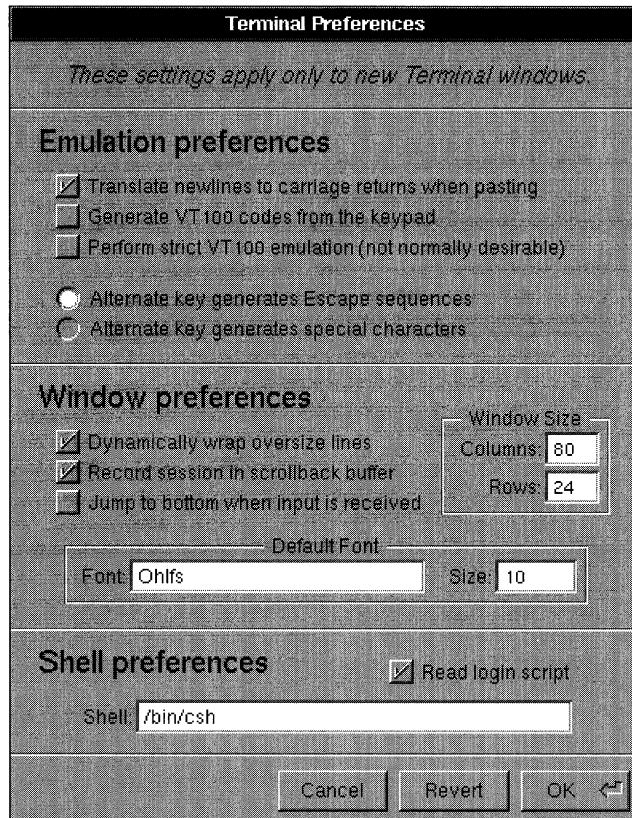


Figure 2-1. The Preferences Panel

Enter values and click buttons to specify new preferences, as described below. Then click OK to set the new preferences (or click Revert to restore the previous settings). New settings remain in effect until you change them. However, Preferences settings only affect new windows—changing settings doesn't affect existing Terminal windows.

Preferences are divided into the following three groups:

- Emulation preferences, which determine what type of terminal emulation a window provides
- Window preferences, which determine a window's size and default font, as well as other characteristics such as line wrap and scrolling
- Shell preferences, which determine which shell or program starts up when a new Terminal window first opens

Emulation Preferences

The following paragraphs describe the emulation preferences that can be set in the Preferences panel.

☑ Translate newlines to carriage returns when pasting

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

☐ Generate VT100 codes from the keypad

If the "Generate VT100 codes from the keypad" box 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.

☐ Perform strict VT100 emulation (not normally desirable)

If the "Perform strict VT100 emulation" box 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 only obeyed if the new size would be larger than the old size.
- The "+" key on the numeric keypad generates a "," character.

◉ Alternate key generates Escape sequences
○ Alternate key generates special characters

When the "Alternate key generates Escape sequences" option 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 that 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.

Window Preferences

The following paragraphs describe the window preferences that can be set in the Preferences panel.

☑ Dynamically wrap oversize lines

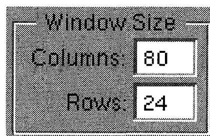
If the “Dynamically wrap oversize lines” box is checked, characters that would extend beyond the right edge of the window wrap around to the beginning of the following line; otherwise, each line of text occupies only one line in the window—the last character that fits on a line gets overwritten by subsequent characters that appear on that line.

☑ Record session in scrollback buffer

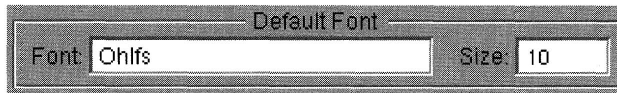
If the “Record session in scrollback buffer” box is checked, windows retain text that scrolls off the top of the window in a scrollback buffer; text that’s scrolled off the window can be scrolled back into view, copied, or printed. Otherwise, text that scrolls off the top of the window can’t be retrieved. Although it’s often useful, scrollback 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).

☐ Jump to bottom when input is received

If the “Jump to bottom 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.



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



Use the Font and Size fields to specify a default font for Terminal windows as follows:

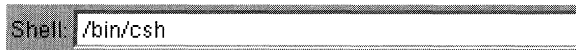
- In the Font field, enter any *fixed-width* font listed in the Font panel. You must enter the font name as it appears in the Font panel—for example, **Ohlfs** or **Courier**. To specify a Courier typeface as well (Ohlfs has no typeface variants), join it to the font name with a hyphen—for example, **Courier-Bold**.

- In the Size field, enter the font size in points.

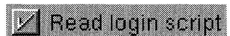
Once you save these settings, new Terminal windows will display text in the specified font.

Shell Preferences

The following paragraphs describe the shell preferences that can be set in the Preferences panel.



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**. The program name you specify is displayed in the title bar of new Terminal windows you open.



If the "Read login script" box is checked, Terminal runs your **.login** file for each new Terminal window you open; otherwise, the **.login** file is ignored.

The Main Menu

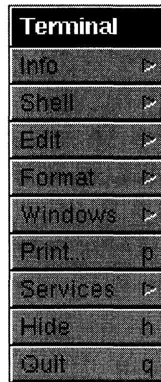


Figure 2-2. The Main Menu

Terminal's main menu contains the commands described below. Some of these commands display submenus related to specific areas of Terminal functionality. These submenus and the commands they contain are described in the sections that follow.

Info

The Info command displays a menu of commands for opening the Info panel and the Preferences panel. The Info panel contains information about the Terminal program, including the version number and copyright information. The Preferences panel is used to set Emulation, Window, and Shell preferences as described earlier in the section "Setting Preferences."

Shell

The Shell command displays a menu that contains the New command for creating a new shell window. This menu also contains a few other commands, described in the section "The Shell Menu."

Edit

The Edit command displays a menu that contains commands for editing text. These are the standard Edit commands, with the exception of Clear Scrollback. The Clear Scrollback command removes all previously displayed text from the main window, except for the current command line.

Format

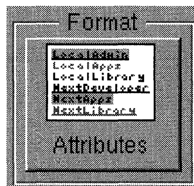
The Format command contains the standard Format commands, which you can use to affect the format of the text in the main window. This menu also contains the standard Page Layout command, which allows you to change the layout of text on the printed page.

Windows

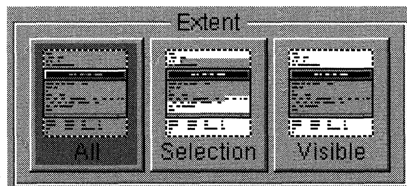
This is the standard Windows command, which displays a menu that contains a list of open Terminal windows. If you choose a window from this list, that window moves to the front and becomes the key window, just as if you had clicked in the window. The menu also contains the standard Miniaturize and Close commands, as well as an Arrange in Front command for bringing all Terminal windows to the front.

Print

The Print command displays a Print panel that you can use to print the contents of the Terminal window. This panel contains the following two options not contained on the standard Print panel.



Click the Format button to toggle between "Attributes" and "No Attributes" mode. "Attributes" indicates that font attributes appear in the printed output; "No Attributes" indicates that the font attributes won't appear.



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

Services

The Services command displays a menu that contains a list of inter-application commands. Selecting one of these commands causes the specified application to become active and provide the requested service.

Hide

The Hide command hides the Terminal program. Terminal's menus and shell windows temporarily disappear from the workspace, but the Terminal program doesn't stop running. When you double-click the Terminal icon, the menus and windows reappear in their previous locations.

Quit

The Quit command causes the Terminal program to stop running.

The Shell Menu

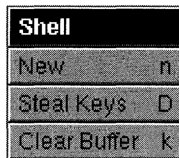


Figure 2-3. The Shell Menu

Terminal's Shell menu provides the following commands.

New

The New command opens a new shell window, using the current Preferences settings.

Steal Keys

The Steal Keys command is useful when you're running the GNU debugger. The purpose of this command is to allow you to effectively debug an application from a shell window in which the debugger is running. The debugging process inevitably 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).

Clear Buffer

The Clear Buffer command removes text from the scrollback buffer, leaving just the current command line.

The Edit Menu

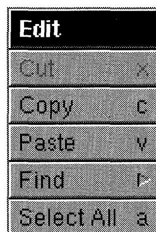


Figure 2-4. The Edit Menu

Terminal's Edit menu provides the standard editing commands.

Cut, Copy, and 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.

The Copy command places a copy of the selected text onto the pasteboard. From the pasteboard, the text can be repeatedly pasted with the Paste command. The pasteboard holds one selection at a time; each new Copy operation overwrites the previous contents of the pasteboard.

Note: The Cut command is always disabled. The only way to remove text from a Terminal window is to use the Clear Buffer command.

Find

The Find command displays a menu that contains commands for finding text, as described later in "The Find Menu."

Select All

The Select All command 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.

The Format Menu

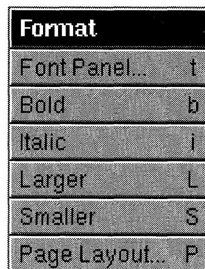


Figure 2-5. The Format Panel

The format menu contains the standard Format 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.

Font Panel

The Font Panel command 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.

Page Layout

The Page Layout command 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.

The Find Menu

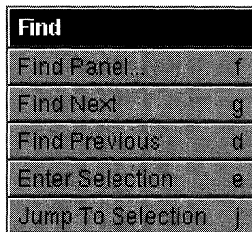


Figure 2-6. The Find Menu

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

Find Panel

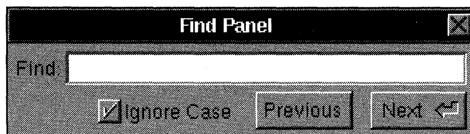


Figure 2-7. The Find Panel

The Find panel allows you to locate the next occurrence of a specified string. In the Find field, enter the string to be located. 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.

Find Next, Find Previous, and Enter Selection

These are the standard Find menu commands described in the *User's Reference Manual*. 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.

The Enter Selection command 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.

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.

Chapter 3
The NeXT Mouse-Based Editor: Edit

3-3 Starting Up Edit

3-4 Opening Edit Files

3-5 Edit Windows

3-6 Selecting Text

3-6 Using the Ruler
3-8 Margins
3-8 Indentation
3-8 Tabs

3-9 Contracting and Expanding Text in a File Window

3-11 Setting Preferences
3-12 Global Options
3-13 Temporary Settings
3-14 Text Options
3-15 C Options

3-17 Keyboard Editing Commands

3-18 Edit and UNIX
3-18 Piping UNIX Output to a File
3-19 Using a Tags File

3-20 The Main Menu

3-20 The File Menu
3-21 Save and Save As
3-21 Open Selection
3-22 Open Directory
3-22 Manager
3-22 Update Directory

3-23 The Edit Menu
3-23 Undelete
3-23 Copy PS
3-24 Nest and Unnest
3-24 Match
3-24 Templates
3-26 Find
3-26 Spelling
3-27 Check Spelling

3-28 The Find Menu
3-28 Find Panel
3-30 Jump to Selection
3-30 Line Range

3-31 The Format Menu
3-31 Structure
3-31 Page Layout

3-32 The Font Menu

3-32 The Text Menu
3-33 Align Left, Center, and Align Right
3-33 Make Rich Text
3-33 Show Ruler or Hide Ruler
3-33 Copy Ruler and Paste Ruler

3-34 The Structure Menu
3-34 Contract All and Expand All
3-34 Contract Sel and Expand Sel

3-34 The Utilities Menu
3-35 Command and User Commands
3-35 Pipe and User Pipes
3-36 Source
3-36 Manual

Chapter 3
The NeXT Mouse-Based Editor: Edit

In addition to the standard UNIX editing tools (**vi**, **ex**, **ed**, and GNU Emacs), NeXT provides a mouse-based text editor named Edit for creating and editing ASCII or RTF (Rich Text Format®) text files. Edit is similar to NeXT's WriteNow® word processor in that it 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. It lacks many of the capabilities found in WriteNow (for example, you can't insert graphics), but it has many features designed for programmers. For example, Edit supports name expansion, directory browsing, block nesting in program listings, a direct interface with the UNIX shell, and more.

Starting Up Edit

You can start up 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. The 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 directory will be searched. See the description of using **tags** files under "Edit and UNIX" below 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 screen coordinates from the left edge of the screen to the left edge of the main menu.
NXMenuY	Specifies the (positive) distance in screen coordinates 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 command, which is described later. Most defaults set with the Preferences command remain in effect until you change them.

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 directory. For example, the following

command would open all the files in the current directory that end with a ".c" extension, plus all the files in a subdirectory 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.

Edit Windows

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

Note: Unless otherwise specified, directory windows mentioned in this chapter are Edit directory windows, not Workspace Manager directory 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, as shown in Figure 3-1.

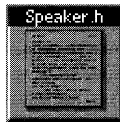


Figure 3-1. Edit Miniwindow

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

Two special features are available in Edit directory 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 directory window.

- You can double-click a file or directory name to open an Edit window displaying that file or directory. This is equivalent to selecting the name and choosing the Open Selection command in the File menu.

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

Selecting Text

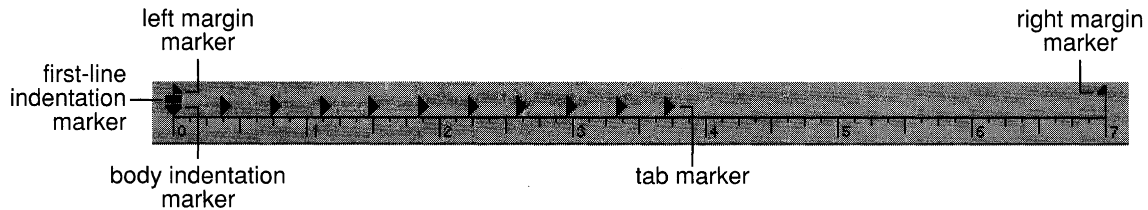
Most operations in Edit are performed on the current selection, which appears either as the insertion point (a blinking vertical bar) or as highlighted text.

You make selections using the standard selecting techniques: You position the insertion point by clicking, and you 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.

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. Edit's ruler is similar to WriteNow's ruler, but it has fewer features, as described in this section. 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 only enabled 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.

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 3-2 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.

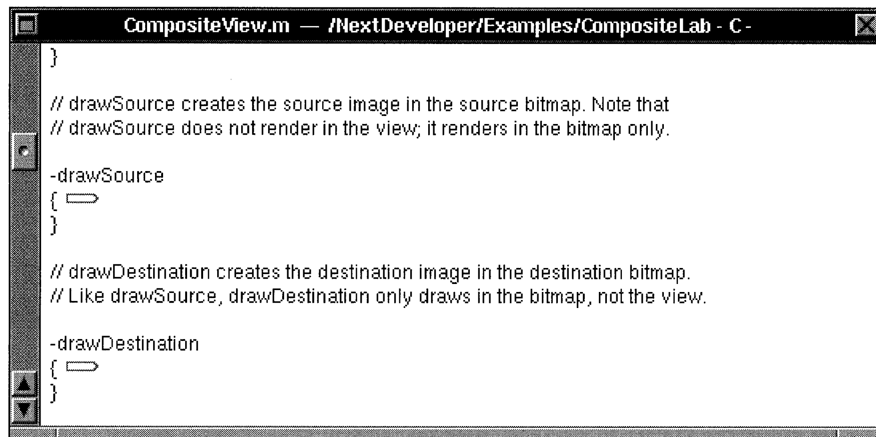


Figure 3-2. File Window with Just First-Level Text Expanded

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 can only affect 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 3-3 shows what the **drawSource** definition looks like after Control-clicking the first of the two text arrows shown in Figure 3-2. 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.

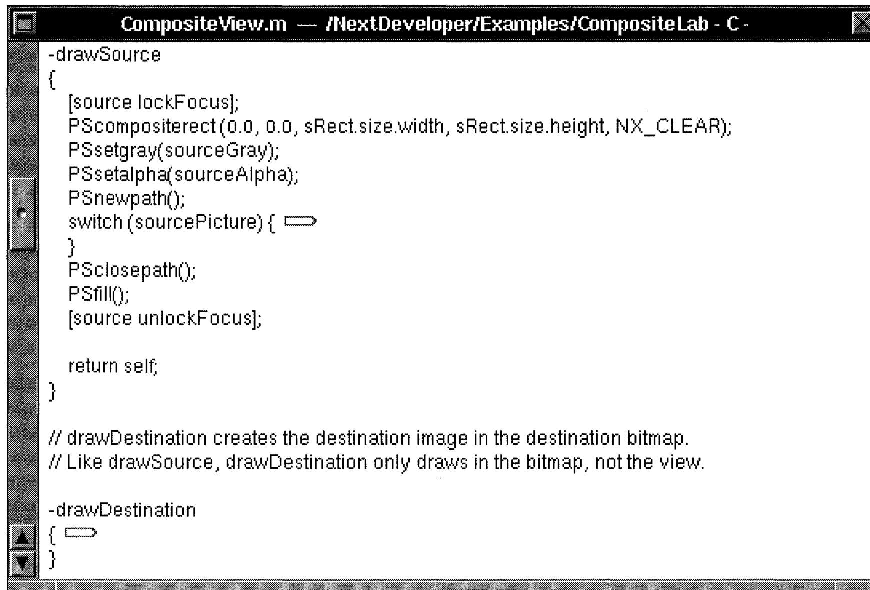


Figure 3-3. File Window with Some Second-Level Text Expanded

Figure 3-4 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.

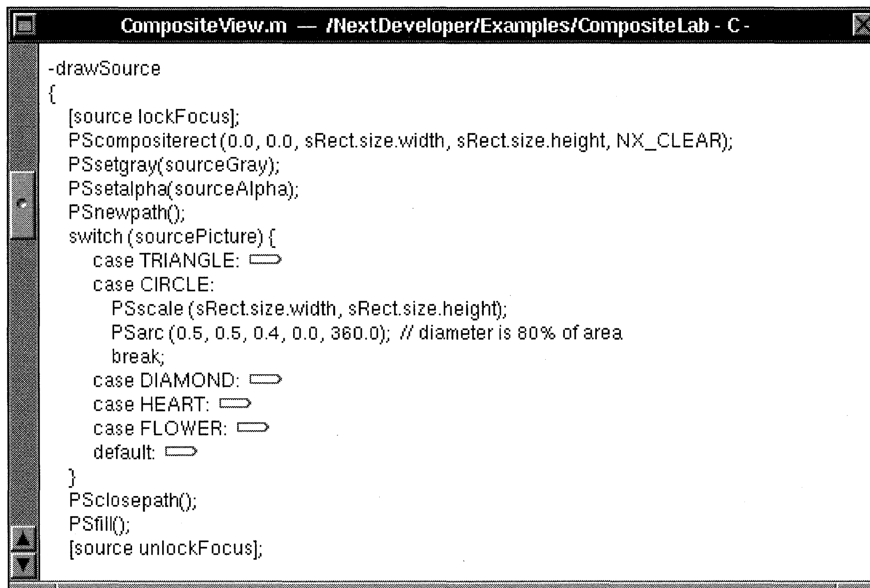


Figure 3-4. File Window with Some Third-Level Text Expanded

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.

Setting Preferences

The Preferences command in the Info menu displays the Preferences panel, shown in Figure 3-5. 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.

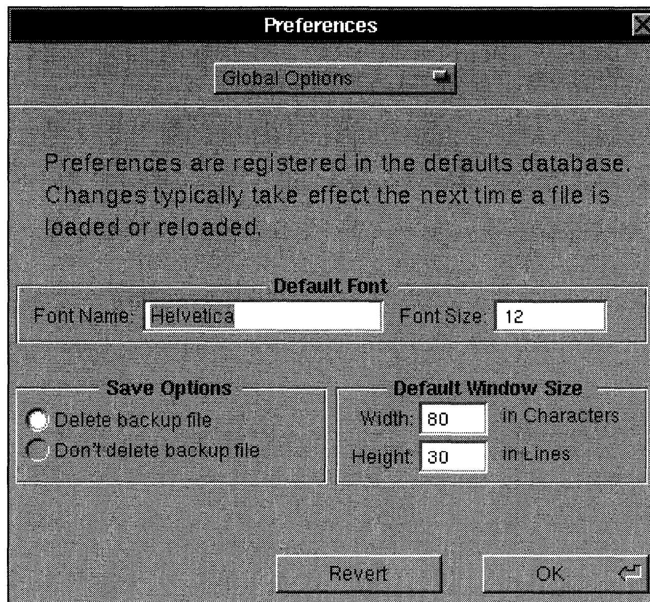
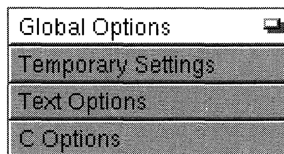


Figure 3-5. The Preferences Panel

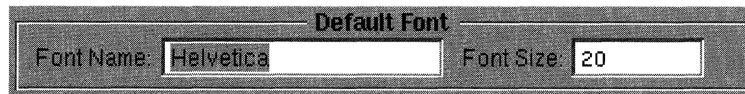
Enter values and click buttons to specify new preferences, as described below. Then click OK to set the new preferences (or click Revert to restore the previous settings). In general, the new settings remain in effect until you change them; however, you can temporarily override some of the defaults by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



In addition to the global options shown in Figure 3-5, you can press the button labeled "Global Options" and in the list that appears, choose from several other sets of options that are available. These additional sets of options are described below, after the global options.

Global Options

The following paragraphs describe the global options that can be set in the Preferences panel.



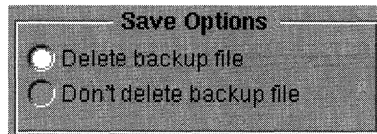
Use the Font Name and Font Size fields to specify a default font for Edit windows as follows:

- In the Font Name field, enter any font listed in the Font panel. You must enter the font name exactly as it appears in the Font panel—for example, **Helvetica**, **Courier-Bold**, or **Times-Roman**.

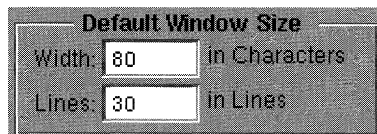
- In the Font Size field, enter the size in points.

After you save these settings, all subsequently opened Edit windows (except those containing RTF files) will display text in the specified font. RTF files open displaying the fonts that were saved with them.

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



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.



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 Lines 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.)

Temporary Settings

Figure 3-6 shows the temporary settings that can be specified in the Preferences panel. These are called temporary settings because they're not saved in your defaults database.

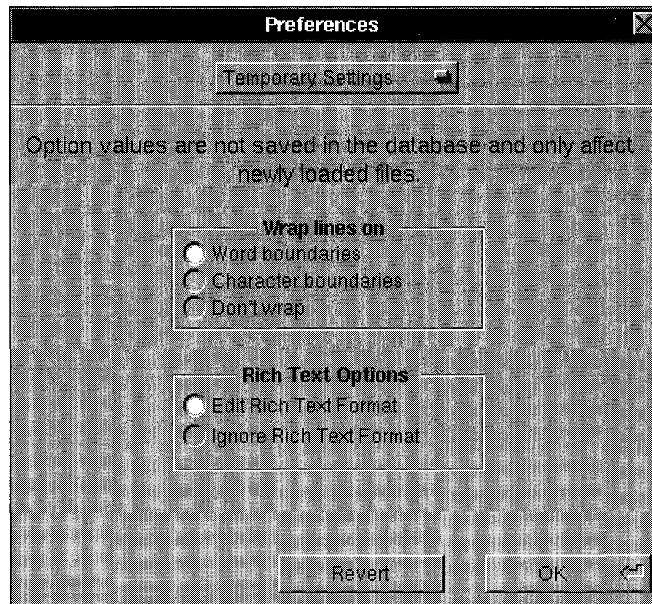
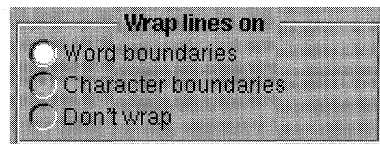
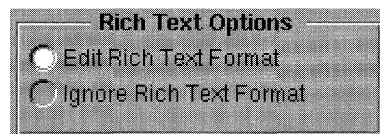


Figure 3-6. Temporary Settings in the Preferences Panel

The following paragraphs describe these temporary settings.



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.



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

Figure 3-7 shows the text options that can be specified with the Preferences panel. Like global settings, text settings are saved in your defaults database and continue to be used until you specify different values for them.

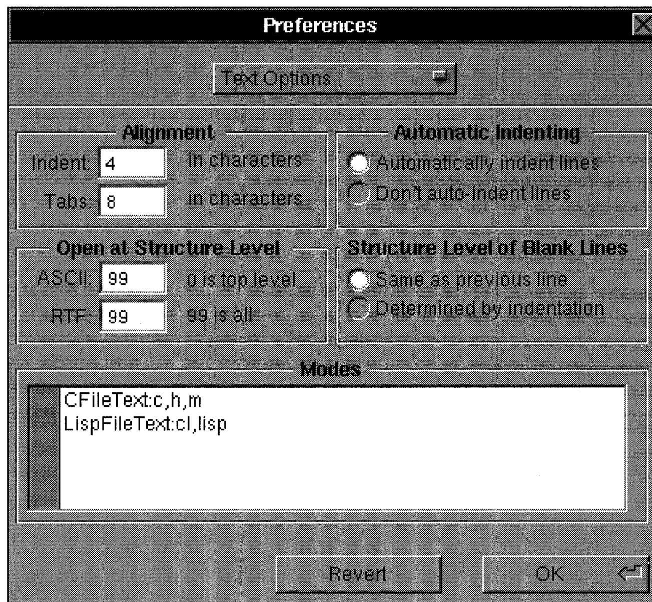
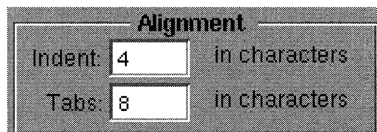
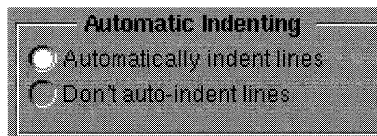


Figure 3-7. Text Options in the Preferences Panel

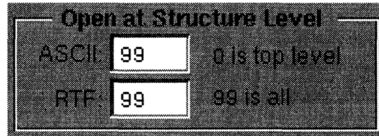
These text options are described in the following paragraphs.



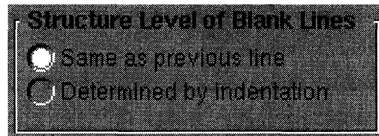
In the Indent field, enter the number of characters you want to shift right or left with the Edit menu's Nest and Unnest commands. In the Tabs field, enter the number of characters you want between tab stops.



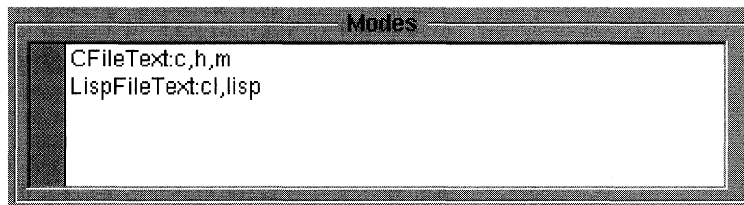
When the "Automatically indent lines" option is selected, Edit indents each new line the same as the line above it (automatic indentation is useful for typing indented lines of code). Click "Don't auto-indent lines" if you want each new line to start at the left margin.



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.



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.



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

Figure 3-8 shows the options that can be specified for files that contain C source code. These settings are saved in your defaults database and continue to be used until you specify different values for them.

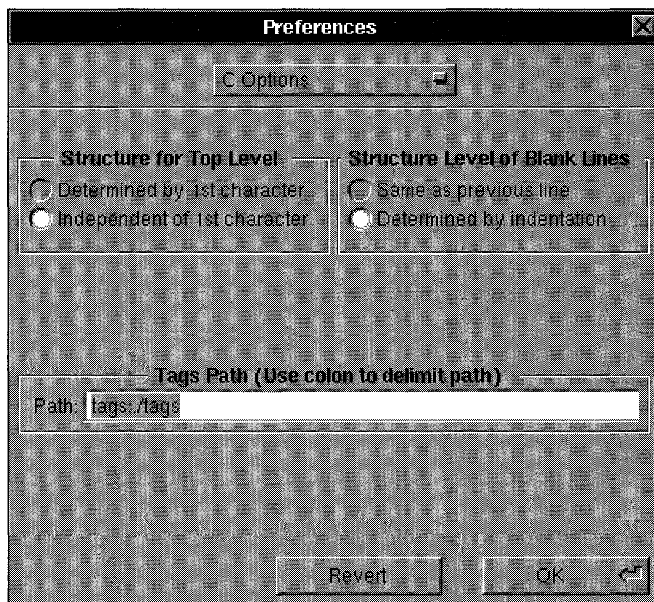
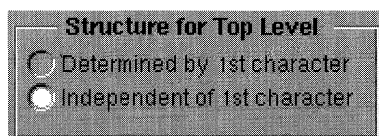
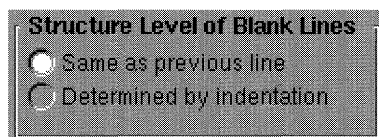


Figure 3-8. C Options in the Preferences Panel

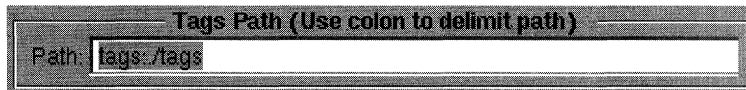
C source code options are described in the following paragraphs.



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.



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.



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 directory (the directory containing the file in the main window) and in the current directory's parent directory. 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 "The Utilities Menu" section later in this chapter for more information about using Edit's Source command with **tags** files.

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

Edit and UNIX

Edit provides some useful commands for using UNIX utilities from within Edit. These commands 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 1990 calendar in an empty window, choose Command in the Utilities menu, enter

```
cal 1990
```

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 directory.

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

command name<tab>*command definition*

For example, the following entry defines a menu item 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 directory 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 directory (the directory 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 "The Utilities Menu" section later in this chapter.

The Main Menu

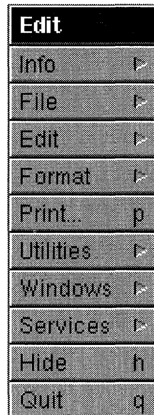


Figure 3-9. The Main Menu

Edit's main menu contains the standard Info, Print, Windows, Services, Hide, and Quit commands described in the *NeXT User's Reference* manual. 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.

The File Menu

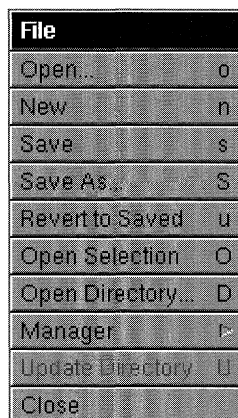


Figure 3-10. The File Menu

Edit's File menu contains the standard Open, New, Revert to Saved, and Close commands, as described in *User's Reference*. The other commands are described in the sections that follow.

Save and Save As

These are the standard Save and Save As commands for saving the contents of the main window on the disk (as described in *User's Reference*).

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 with the Preferences command.

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

The Open Selection command opens the file or directory currently selected in the main window. Normally, you use this command on a selection in a directory 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 directory (the directory containing the file in the main window).

Open Directory

The Open Directory command displays a panel (shown in Figure 3-11) in which you enter the pathname of a directory to be opened; when you click OK, the directory opens in an Edit directory window. When the panel appears, Edit displays the name of the current directory in the "Directory name" field.

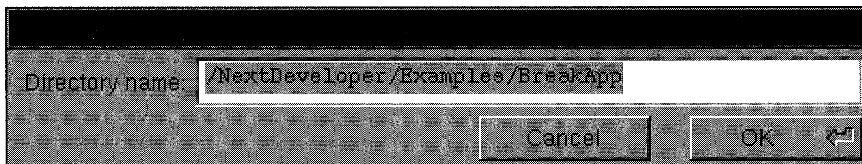


Figure 3-11. The Open Directory Panel

Manager

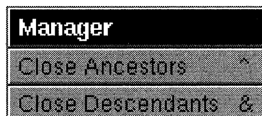


Figure 3-12. The Manager Menu

The Manager command opens a menu that contains two commands for managing which Edit windows remain open in the workspace. These commands are especially useful for working with different versions of program code.

- The Close Ancestors command closes all Edit windows associated with each directory that's neither the main window's directory nor one of its subdirectories.
- The Close Descendants command closes all Edit windows associated with each directory that's a subdirectory of the main window's directory. If the main window is a directory window it will remain open, but if the main window is a file window it will be closed.

Update Directory

The Update Directory command updates the contents of the main window, which must be a directory window. Directory windows aren't automatically updated, so this command is useful when files in a directory have been created, deleted, or renamed.

The Edit Menu

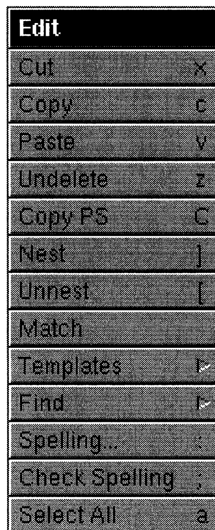


Figure 3-13. The Edit Menu

Edit's Edit submenu contains the standard Cut, Copy, Paste, and Select All commands, as described in *User's Reference*. The other commands and the submenus they open are described in the sections that follow.

Undelete

The Undelete command 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.

Copy PS

The Copy PS command copies the entire contents of the main window (*not* just the current selection) onto the pasteboard as an Encapsulated PostScript (EPS) image. The text in the main window is unaffected; however, once pasted into an application that accepts EPS images, the copy of the text can no longer be edited.

Nest and Unnest

These commands 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 with the Preferences command or overridden the default when you started up Edit from a shell window).

The Unnest command moves the selected lines the default number of characters to the left; it thus counteracts the effect of the Nest command.

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 simply by double-clicking either of the delimiters.

Templates

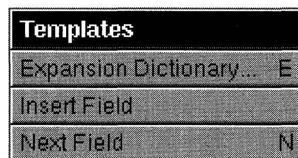


Figure 3-14. The Templates Menu

The Templates command displays a menu of commands for creating and using glossary entries—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 glossary entries, choose the Expansion Dictionary command in the Templates menu to open the window shown in Figure 3-15.

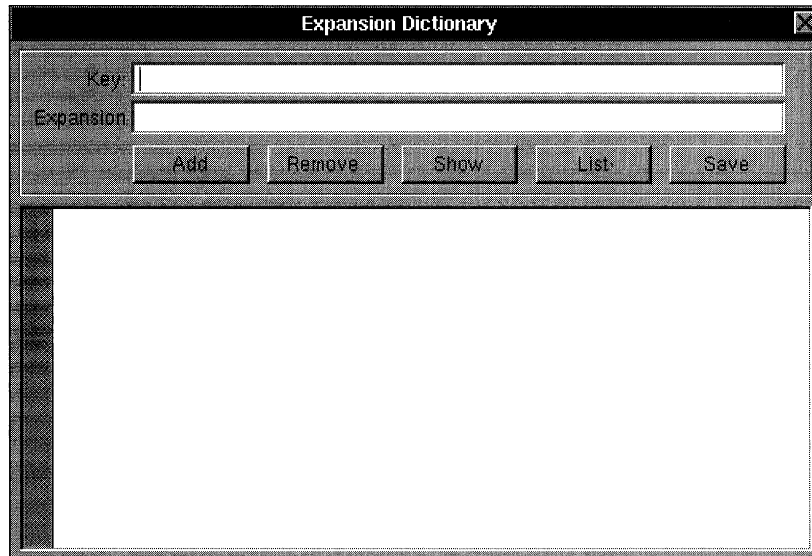


Figure 3-15. The Expansion Dictionary Window

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 Templates 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:

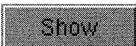
```
Subject:  «subject»
To:  «recipient»
cc:  «cc»»

«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.

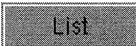
After entering the abbreviation and the expanded text it stands for in the Key and Expansion fields, click the Add button to accept the new glossary entry.

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

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



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

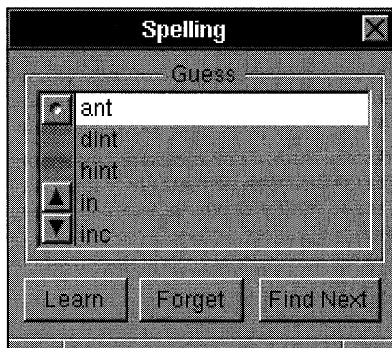


Click List to view a list of all available glossary entries.

Find

The Find command displays a menu containing standard commands for finding or replacing text as well as a command for searching for lines or characters by number. See "The Find Menu" later in the chapter for more information.

Spelling



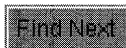
The Spelling command brings up a panel that lets you check the spelling of words, choose from possible corrections, and modify the spelling dictionary. As a convenience, Edit doesn't bring up the Spelling panel as the key window, so that you can type to correct a misspelling without having to click in 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 system-wide 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).

Check Spelling

Choosing the Check Spelling command has the same effect as clicking Find Next in the Spelling panel (see the Spelling command above). Spelling locates and selects the next word not contained in the spelling dictionary. To replace the misspelled word, you can just begin typing.

For more options when checking spelling, use the Spelling command, either after Check Spelling or instead of Check Spelling.

The Find Menu

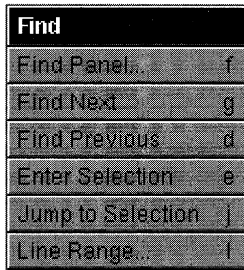


Figure 3-16. The Find Menu

Edit's Find menu contains the standard Find Next, Find Previous, and Enter Selection commands described in *User's Reference*. The other commands are described in the sections that follow.

Find Panel

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.

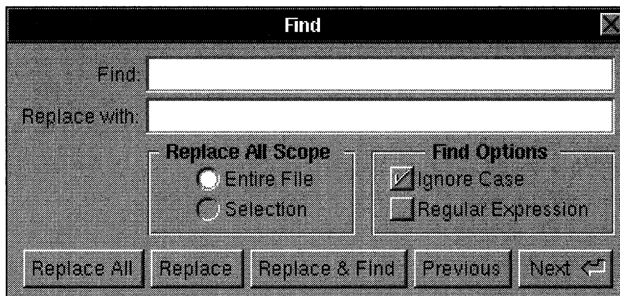
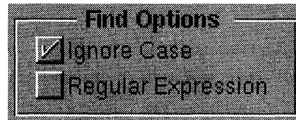


Figure 3-17. The Find Panel

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.

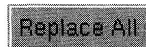
In the "Replace with" field, you may specify a replacement string. Then click one of the panel's buttons to perform the exact search operation you want, as described below. If the end of the document is reached during a search, Edit continues searching from the beginning of the document; when searching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



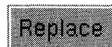
When Ignore Case is checked, Edit doesn't distinguish between capital and lowercase letters when finding a match during the search. If Ignore Case is not checked, the search is case-sensitive.

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.

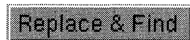
The Replace All Scope options specify whether Replace All applies to the entire document (Entire File) or only to the current text selection (Selection).



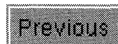
In the area that you specify, the Replace All button replaces all occurrences of the text entered in the Find field with the text entered in the "Replace with" field. If the "Replace with" field is blank, Replace All deletes all occurrences of the text. After a search with Replace All, the Find panel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

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.

Line Range

The Line Range command opens the Line and Character Range panel, which identifies by number the line or line range containing the current selection in the main window. If the Character option is selected instead of the Line option, then the character range is displayed instead of the line range.

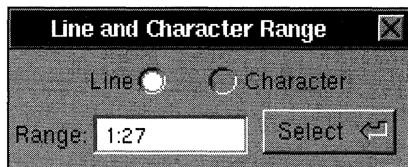


Figure 3-18. The Line and Character Range Panel

You can also use this panel to search by number 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.

The Format Menu

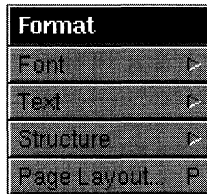


Figure 3-19. The Format Menu

The Format menu contains commands for displaying the standard Font and Text commands described in *User's Reference*. These menus also contain some nonstandard commands, which are described later in this chapter.

Structure

The Structure command opens a menu that contains commands for contrasting and expanding text, as described in the section "Contracting and Expanding Text in a File Window."

Page Layout

The Page Layout command displays the standard Page Layout panel for choosing among various paper sizes, scaling factors, and orientations for text printed from the main window, as described in *User's Reference*.

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.

The Font Menu

Font
Font Panel... t
Bold b
Italic i
Underline
Larger
Smaller
Heavier
Lighter
Superscript /
Subscript \
Unscript
Light Gray
Dark Gray
Black
Copy Font 3
Paste Font 4

Figure 3-20. The Font Menu

The Font menu contains the standard Font commands described in *User's Reference*, plus a few additional commands that let you change the font properties of the text displayed in the main window. If you've changed the file to RTF (using the Make Rich Text command in the Text menu), font changes apply to the current selection and are saved when you save the contents of the window. If the file isn't an RTF 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.

The Text Menu

Text
Align Left {
Center -
Align Right }
Make Rich Text R
Show Ruler r
Copy Ruler 1
Paste Ruler 2

Figure 3-21. 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 text in the main window from ASCII to RTF.

Align Left, Center, and Align Right

These commands 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

The Make Rich Text command changes the format of the text in the main window from ASCII to RTF. Once the text is converted to RTF, 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. Once you've converted the file to RTF, you can convert it back to ASCII by copying the text, pasting it into a new non-RTF window, and saving the contents of the new window.

Show Ruler or 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" earlier in the chapter for a detailed introduction to this Edit feature.

Copy Ruler and Paste Ruler

The Copy Ruler command 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 the Paste Ruler command. It's as though there's a separate pasteboard for the ruler, and Copy Ruler replaces what's already on it, just as the Copy command does for text.

The Paste Ruler command 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 the Copy Ruler command. If the current selection spans more than one paragraph, Paste Ruler replaces the ruler settings of all the selected paragraphs.

Neither of these commands requires the ruler to be showing, or changes the contents of the pasteboard.

The Structure Menu

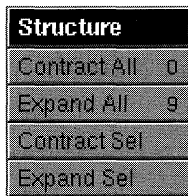


Figure 3-22. The Structure Menu

The Structure command displays a menu of 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 multi-level 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.

Contract All and Expand All

The Contract All and Expand All commands contract or expand all the text in the main window.

Contract Sel and Expand Sel

The Contract Sel and Expand Sel commands contract or expand just the selected text in the main window.

The Utilities Menu

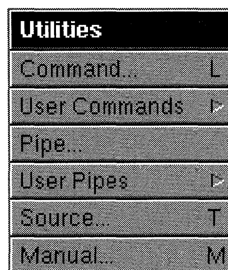


Figure 3-23. The Utilities Menu

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, to which you can add commands that you've defined yourself.

Command and User Commands

The Command 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 This refers to the file that's displayed in the main window.

$selection This 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 directory window.

The User Commands command displays a menu of commands you've defined and saved in a file named **.commanddict** in your home directory. 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 the section "Piping UNIX Output to a File" earlier in this chapter.

Pipe and User Pipes

The Pipe command works the same as the User 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.

The User Pipes command displays a menu that contains pipe commands you've defined and saved in a file named **.pipedict** in your home directory. 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 in the section "Piping UNIX Output to a File" earlier in this chapter.

Source

The Source command 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 directory (the directory 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

The Manual command displays a UNIX manual page in an Edit window. First select the manual page subject in the main window and then choose the Manual command. If there's no selection, a panel appears prompting you for an entry.

Chapter 4
Developer Applications and Utilities

4-3 The Object Browser Application: AppInspector
4-4 Opening an Application: The Select Panel
4-4 Browsing Instances and Classes: The Browser Panel
4-5 Browsing Instances: The Instance Browser
4-7 Browsing Classes: The Class Browser
4-7 Inspecting Instances and Classes: The Inspector Panel
4-8 Inspecting Instances: The Instance Inspector
4-9 Inspecting Memory: The Memory Inspector
4-10 Inspecting Classes: The Class Inspector
4-11 Finding Classes, Methods, and Variables: The Find Panel
4-12 Run-Time Tracing: The Peep Window
4-13 The Main Menu
4-13 The Application Menu
4-14 The Tools Menu
4-14 The Browse Menu

4-15 The Malloc Debugger Application: MallocDebug
4-15 Preparing Your Application
4-16 Using MallocDebug
4-18 Identifying Damaged Nodes
4-18 Finding Memory Leaks
4-18 Measuring Memory Usage
4-19 The Main Menu
4-19 The Application Menu

4-20 The Process Monitoring Application: ProcessMonitor
4-20 Selecting a Process: The Processes Panel
4-21 Inspecting a Process: The Inspector Panel
4-22 The Control Inspector
4-22 The Mach Inspector
4-23 The Display PostScript Inspector
4-24 The Malloc Inspector
4-24 The Objective-C Inspector
4-25 Monitoring Memory Usage: The Mach Monitor
4-25 The Main Menu
4-26 The Processes Menu
4-26 The Monitor Menu

4-27 NeXT's PostScript Window Server Interface: pft

4-27 Starting the **pft** Program

4-28 Executing PostScript Code from a File

4-28 Setting Up a Window

4-29 Flushing the Server's Output Buffer

4-30 Summary Example

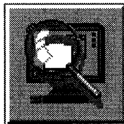
Chapter 4
Developer Applications and Utilities

This chapter describes the following miscellaneous applications available for use in programming on NeXT computers:

- AppInspector—look into a running application and examine its data
- MallocDebug—measure the dynamic memory usage of the applications you write
- ProcessMonitor—examine and get information about running processes

These applications are all located in **/NextDeveloper/Apps**, and are described in the following sections. The final section describes **pft**, a shell-based interface to the PostScript Window Server.

The Object Browser Application: AppInspector



AppInspector lets you look inside a running application and examine its data. You can also use it to perform run-time tracing of the object instances in the program. That is, you can choose any instance in the application and see all messages that come to that object. This information is presented in an easy-to-use, graphical way.

AppInspector is run in the standard NeXT way, from the workspace. Once AppInspector is running, an application can be brought up directly from AppInspector, or indirectly from GDB:

- To choose an application from AppInspector, select the Open command from the Application menu. This brings up the Select panel, which contains the icons of all the applications that you are currently running. You can select any of these running applications by clicking its icon.

- If you run GDB on an application, you can type "browse *foo*," where *foo* is an expression that evaluates to an object instance. That expression will be among those being browsed in AppInspector.

Two panels are allocated for each application that is being browsed—a browser panel and an Inspector panel. These panels (and other AppInspector panels and windows shared among browsed applications) are described in the following sections.

Opening an Application: The Select Panel

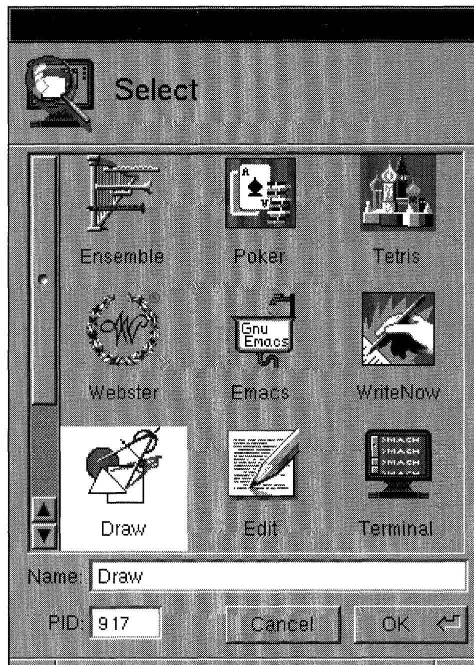


Figure 4-1. The Select Panel

The Select panel is displayed when you choose the Open command from the Application menu. This panel displays the name and icon of each application that's currently running; only applications with an __ICON segment can be opened in AppInspector. When you select the icon of an application and click the OK button, a browser panel for the selected application appears. This browser panel is described in the following section.

Browsing Instances and Classes: The Browser Panel

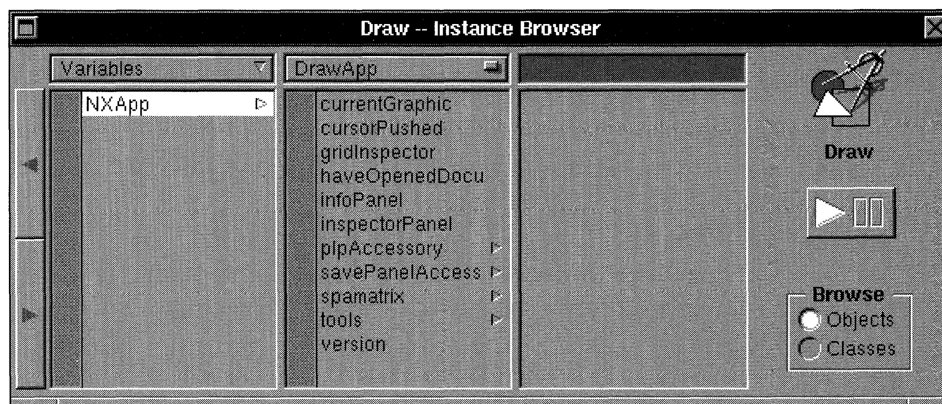
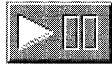
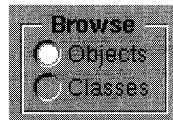


Figure 4-2. The Browser Panel

A browser panel opens automatically when you select an application with the Select panel. The application that you selected with the Select panel is shown in the upper right corner of the browser panel. You can have multiple applications open, each with its own browser panel, so the icon serves to identify which application you're inspecting.



The button immediately under the icon of the browsed application indicates whether the application is currently running or paused. You can pause the application or continue running it by clicking this button.



In the lower right corner of the browser panel is a Browse switch, which lets you alternate between two modes—one mode is for browsing objects (that is, instances), the other is for browsing classes. The browser panel is referred to as either the Instance Browser or the Class Browser, depending on which of the two modes is currently chosen.

Browsing Instances: The Instance Browser

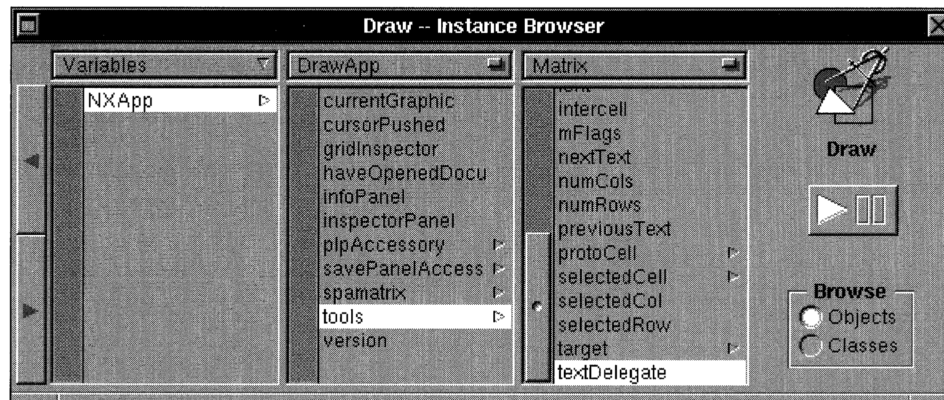
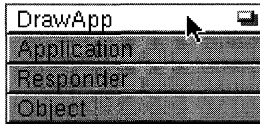


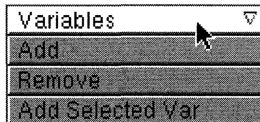
Figure 4-3. The Instance Browser

The Instance Browser appears when you choose the Objects command in the Browse menu. This browser lets you see the instance variables belonging to objects in the application, and get information about the instances that are assigned to those variables.

For example, the highlighted NXApp variable in Figure 4-3 has assigned to it an instance of the class DrawApp, as indicated by the class name DrawApp appearing in the control at the top of the adjacent column—the class name DrawApp indicates that the column contains a list of the instance variables defined by the DrawApp class.



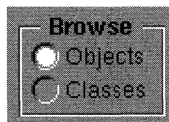
Besides indicating that the DrawApp class defines the instance variables displayed in the column, the control labeled DrawApp also shows the inheritance hierarchy of the DrawApp class when you press the control. Dragging to another class in the inheritance hierarchy causes the column to display the instance variables defined by the selected class, rather than the instance variables defined by the DrawApp class (of course, the control label is renamed accordingly). In this way, you can use the instance browser to see not only the class of the instance assigned to a particular variable, but also the instance variables defined by that class and by all the other classes above it in the inheritance hierarchy.



The instance browser may contain any number of top-level instance variables. The NXApp global variable is the only one that's there by default, but you can add others with the Variables control, located above the leftmost column in the Instance Browser. This control contains three options:

Option	Effect
Add	Displays the Add Object panel, in which you specify the name and address of a variable to be added to the list of top-level variables.
Remove	Removes the currently selected top-level variable from the list of top-level variables.
Add Selected Var	Adds to the list of top-level variables whatever variable is currently selected in the instance browser.

You can also use the "browse" command in GDB to add top-level instance variables to the instance browser.



If you select the Classes option in the Browse switch at the lower right corner of the Instance Browser, the browser panel will change to a Class Browser instead of an Instance Browser (the class that's the type of the currently selected variable will be automatically displayed in the Class Browser). The Class Browser is described in the following section.

Browsing Classes: The Class Browser

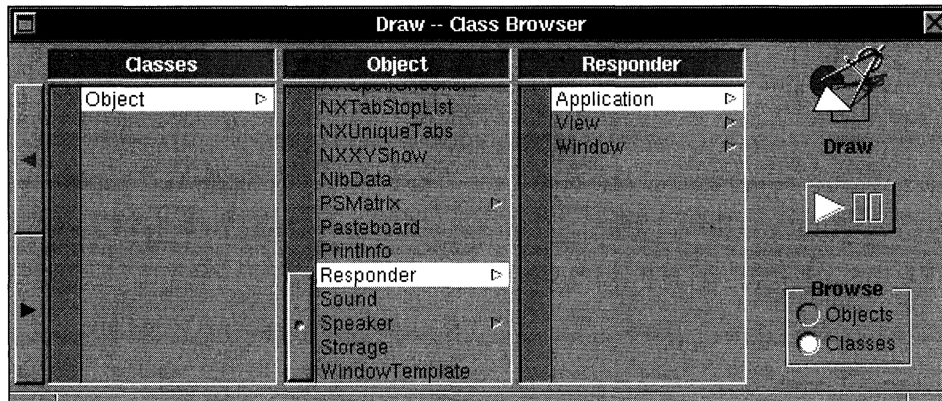


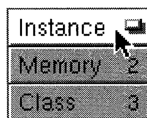
Figure 4-4. The Class Browser

The Class Browser appears when you choose the Classes command in the Browse menu. This browser shows all classes known to the application. You can browse the inheritance hierarchy by selecting classes and subclasses.

Besides showing you the inheritance hierarchy and the relationships among the various classes, the Class Browser is useful mostly in conjunction with the inspector panel, which displays detailed information about whatever class is currently selected in the Class Browser. The following section describes the inspector panel and the information it provides.

Inspecting Instances and Classes: The Inspector Panel

An inspector panel appears when you choose the Inspect command in the Tools menu. This browser shows information about various attributes of the cell that's currently selected in the Instance Browser or Class Browser. Each opened application has its own inspector panel, shown in Figure 4-5.



Each inspector panel has a popup menu that lets you select any of three modes. The inspector panel is referred to as either the Instance Inspector, the Memory Inspector, or the Class Inspector, depending on which of the three modes is currently selected.

Inspecting Instances: The Instance Inspector

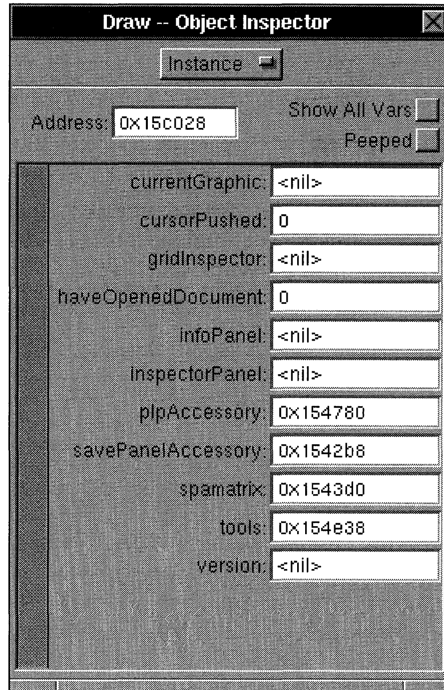


Figure 4-5. The Instance Inspector

The Instance Inspector displays information about the object that's currently selected in the Instance Browser. This information includes the object's address and the values of the instance variables defined by the object.

Show All Vars

If the Show All Vars button is unchecked, only the variables defined by the currently displayed object are shown. Check the Show All Vars button to show inherited variables as well as the variables defined by the object that's being inspected.

Peeped

If you check the Peeped box, the Peep window appears and all messages sent to the currently displayed object are displayed in the Peep window. (This option works only if you have linked your application against **libPeep.a**.)

Inspecting Memory: The Memory Inspector

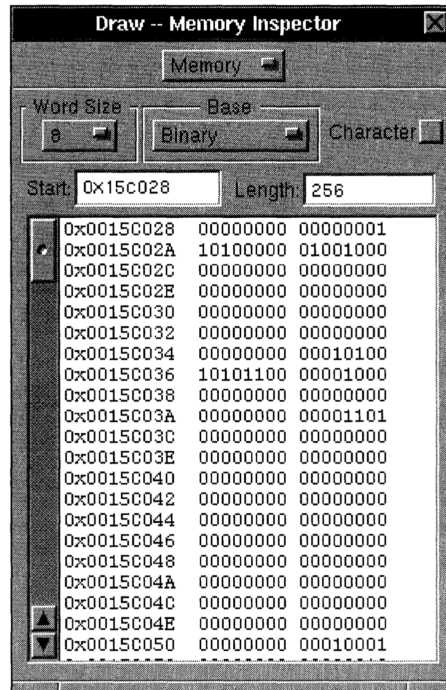
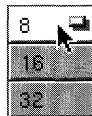
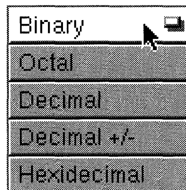


Figure 4-6. The Memory Inspector

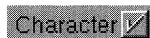
The Memory Inspector displays the contents in memory of the object that's currently selected in the Instance Browser.



Press the Word Size button and drag to change the word size used in the display from 8 bits to 16 or 32 bits.



Press the Base button and drag to change the display from binary to some other base representation.



If the Character box is checked, the memory data is displayed as ASCII characters.

Inspecting Classes: The Class Inspector

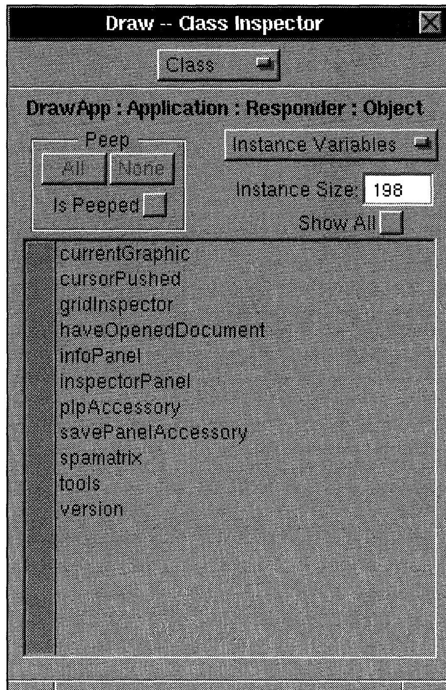
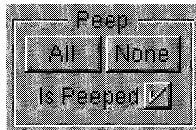
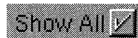


Figure 4-7. The Class Inspector

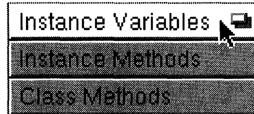
The Class Inspector displays information about the class that's currently selected in the Class Browser.



Check the Is Peeped box to peep the selected method. Click All or None to peep all or none of the displayed variables or methods.



If the Show All box is unchecked, only the instance variables, instance methods, or class methods defined by the currently displayed object are shown. Check the Show All box to show inherited variables or methods as well as the variables or methods defined by the class that's being inspected.



This button lets you display either the instance variables, instance methods, or class methods of the selected class in the Class Inspector. If you choose the Instance Methods option or the Class Methods option, you can specify which methods should be peeped by selecting individual methods, or by clicking the All or None buttons in the Peep box.

Finding Classes, Methods, and Variables: The Find Panel

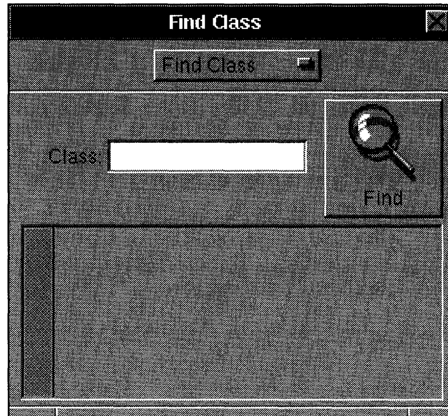
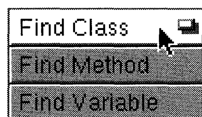


Figure 4-8. The Find Panel

The Find panel is displayed when you choose the Find command from the Tools menu. You can use the Find panel to search for classes, methods, and variables used in the application.



This button lets you specify whether you want to find a class, a method, or a variable. Choose the appropriate option, type the name in the text field of the Find panel (wildcard characters can be used—for example, *init*), and click the Find button. Items that are found are displayed in the lower portion of the Find panel.

Run-Time Tracing: The Peep Window

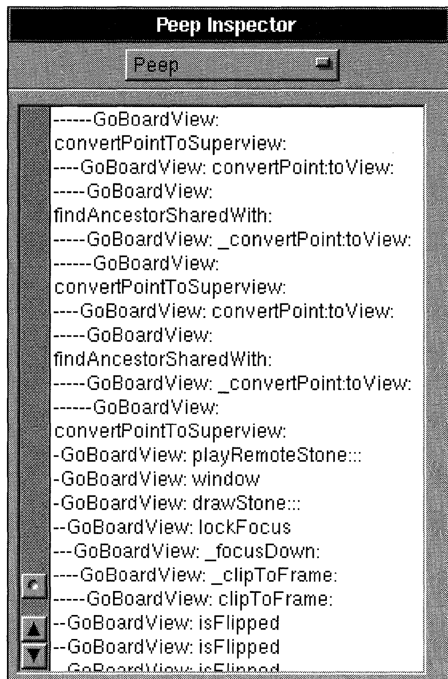


Figure 4-9. The Peep Window

To use the Peep window with an application, you must first link the application with the library **/usr/lib/libPeep.a** by specifying the linker option **-lPeep**. If your application is built with Interface Builder, you can simply add **/usr/lib/libPeep.a** to the "Other libs" section of the Project Inspector.

Once the application has been linked with **libPeep.a**, AppInspector can peep any of the application's instance variables displayed in the Instance Browser. Just bring up the Instance Inspector for that object; then select the Peeped button and click the Pause/Play button. Any messages sent to the instance will be displayed in the Peep window.

Methods can also be selected individually. To do this, select from the Instance Methods inspector the methods you wish to see. If you want to stop tracing, go to the Instance Inspector and click the Peeped button again.

Class methods can be traced by using the Class Browser and selecting the Instance Inspector. There are Peeped buttons for them as well. Select a class, and any class methods sent to that class will be displayed in the Peep window.

The Main Menu

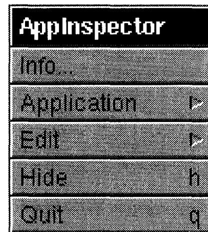


Figure 4-10. The Main Menu

AppInspector's main menu contains the standard Info, Edit, Hide, and Quit commands. The Application command brings up the Application menu, described in the following section.

The Application Menu

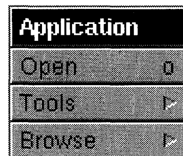


Figure 4-11. The Application Menu

AppInspector's Application menu contains the Open, Tools, and Browse commands:

Command	**Effect**
Open	Displays the Select panel, from which you select and open a running application. See the section "Opening an Application: The Select Panel" for a description of this panel.
Tools	Brings up the Tools menu, which contains commands for displaying the various AppInspector panels.
Browse	Brings up the Browse menu, which contains commands for browsing instances and classes.

The Tools Menu

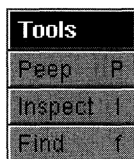


Figure 4-12. The Tools Menu

The Tools menu contains the Peep, Inspect, and Find commands:

Command	**Effect**
Peep	Brings up the Peep panel, which displays messages sent to any peeped object displayed in the Instance Browser. See the section "Run-Time Tracing: The Peep Window" for details.
Inspect	Brings up the Inspector panel, described in the section "Inspecting Instances and Classes: The Inspector Panel."
Find	Brings up the Find panel, which you can use to search for a class, a method, or a variable. See the section "Finding Classes, Methods, and Variables: The Find Panel" for details.

The Browse Menu

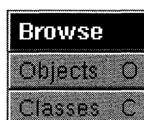
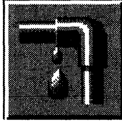


Figure 4-13. The Browse Menu

The Browse menu contains the Objects and Classes commands:

Command	**Effect**
Objects	Brings up the Instance Browser, described in the section "Browsing Instances: The Instance Browser."
Classes	Brings up the Class Browser, described in the section "Browsing Classes: The Class Browser."

The Malloc Debugger Application: MallocDebug



MallocDebug measures the dynamic memory usage of applications. You can use MallocDebug to measure all allocated memory in an application, or to measure the memory allocated since a given point in time. MallocDebug also contains 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 MallocDebug, you must first link your application with a library containing a special version of **malloc** that can communicate with MallocDebug. 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

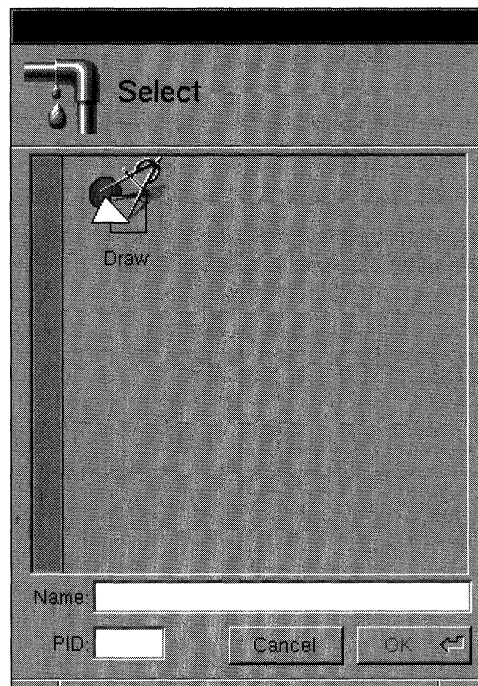


Figure 4-14. The Select Panel

To use MallocDebug, you must first select an application to monitor. Choose the Open command in the Application menu to bring up the Select panel. Only 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.

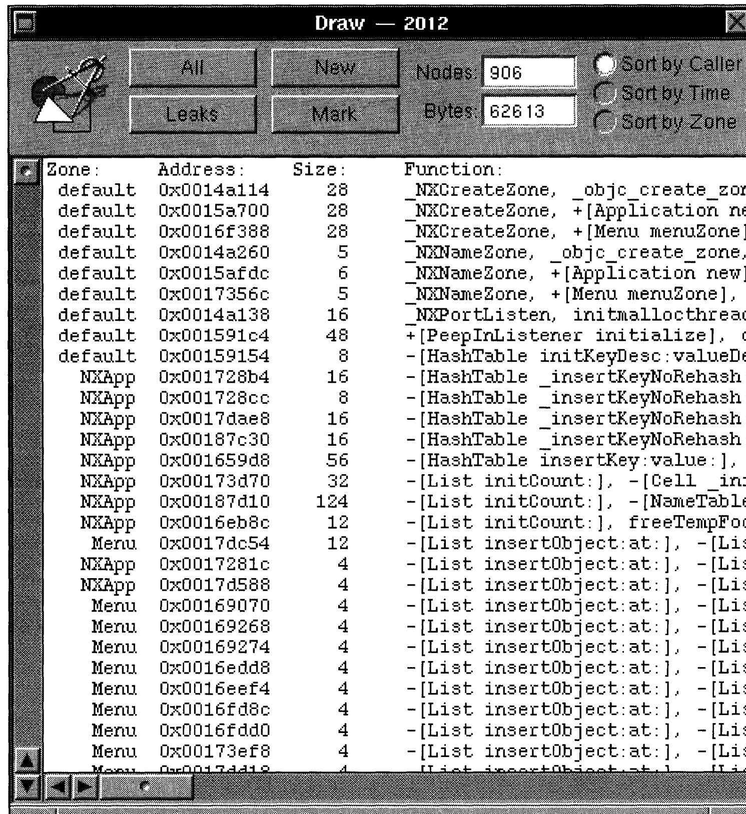
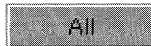
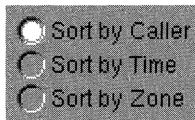


Figure 4-15. The Application Panel

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 one of NeXT's zone allocation functions (**NXZoneMalloc**, **NXZoneRealloc**, **NXZoneCalloc**). As shown in Figure 4-15, 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.

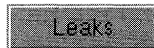
Identifying Damaged Nodes

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.



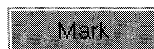
When you click the Leaks button, MallocDebug searches through your program's memory for pointers to each node. Any node that can't be referenced is displayed as a memory leak.

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.

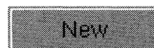
Note: The garbage detector only searches for references to the beginning of each node. If your program doesn't retain a pointer to the start of a node, but instead retains a pointer into the middle of it, that node will show up as a leak even though it really isn't one.

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.

The Main Menu

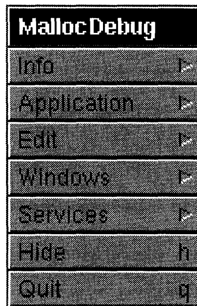


Figure 4-16. The Main Menu

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

The Application Menu

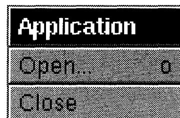
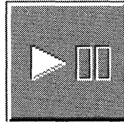


Figure 4-17. The Application Menu

MallocDebug's Application menu contains the Open and Close commands for opening and closing applications:

Command	**Effect**
Open	Displays the Select panel, from which you select and open a running application. See the section "Opening an Application: The Select Panel" for a description of this panel.
Close	Closes the Application panel for the selected application; the debugged application remains running.

The Process Monitoring Application: ProcessMonitor



ProcessMonitor can be used to examine any running process. ProcessMonitor 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®, Malloc, and the run-time environment.

Selecting a Process: The Processes Panel

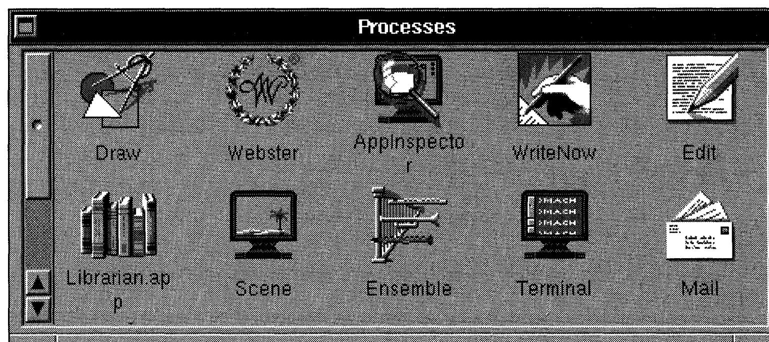


Figure 4-18. The Processes Panel

When ProcessMonitor 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 which 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

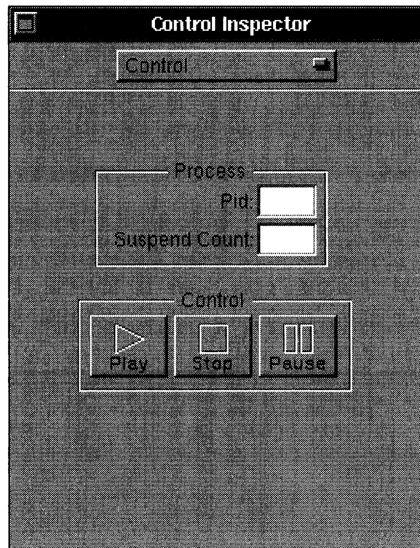
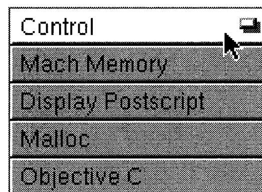


Figure 4-19. The Inspector Panel

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

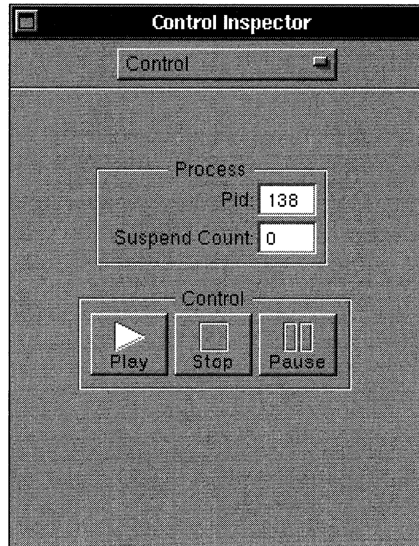


Figure 4-20. The Control Inspector

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

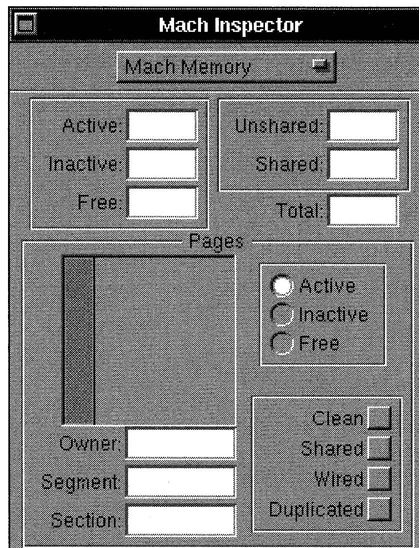


Figure 4-21. The Mach Inspector

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 brings up the Mach Monitor panel, which can be used to provide a dynamic record of Mach memory usage. For more information, see the section "Monitoring Memory Usage: The Mach Monitor."

The Display PostScript Inspector

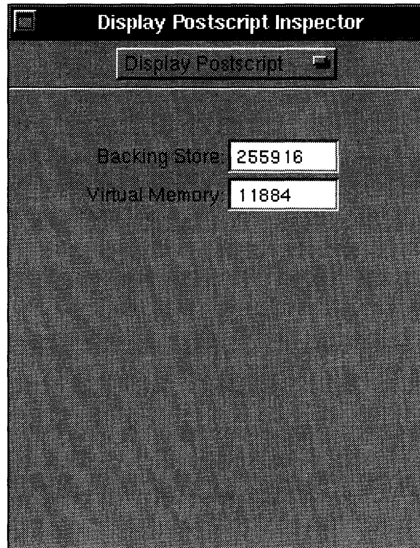


Figure 4-22. The Display PostScript Inspector

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

The Malloc Inspector

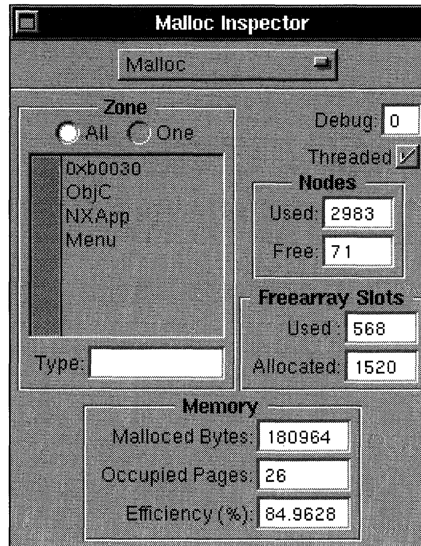


Figure 4-23. The Malloc Inspector

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

The Objective-C Inspector

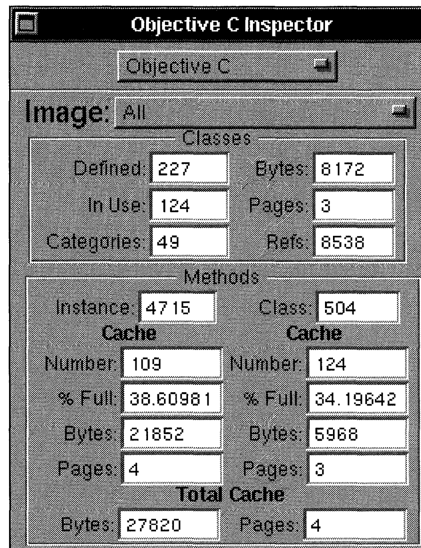


Figure 4-24. The Objective-C Inspector

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

Monitoring Memory Usage: The Mach Monitor

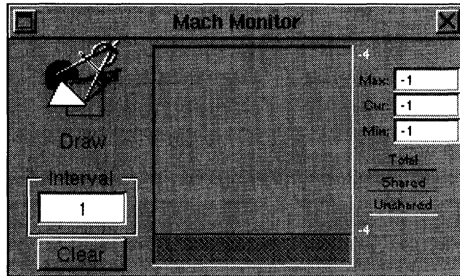


Figure 4-25. The Mach Monitor Panel

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.

The Main Menu

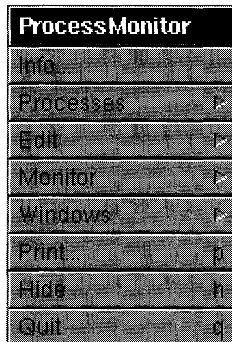


Figure 4-26. The Main Menu

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

The Processes Menu

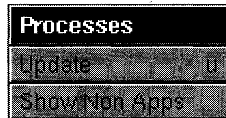


Figure 4-27. The Processes Menu

ProcessMonitor's Processes menu contains the Update and Show Non Apps commands:

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	Causes the Processes panel to show all processes, not just those processes associated with NeXTstep® applications. Choosing this command (now labeled Hide Non Apps) a second time causes the panel to show just application processes again.

The Monitor Menu

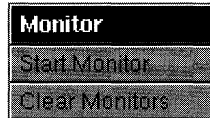


Figure 4-28. The Monitor Menu

ProcessMonitor's Monitor menu contains the Start Monitor and Clear Monitors commands:

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 Panel"). This command is only enabled when the Mach Inspector or the Display Postscript Inspector is open.
Clear Monitors	Clears the contents of the Mach Monitor panels.

NeXT's PostScript Window Server Interface: pft

pft is a simple shell-based utility for communicating with the PostScript Window Server. You start up the **pft** program by entering "pft" 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 also, the file name must be an absolute pathname:

```
(~/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 NeXT 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 the following:

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 the following:

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 output 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 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
```

Chapter 5
The GNU C Compiler

5-3 GNU CC Command Options
5-4 Global Compilation Options
5-5 C Preprocessor Options
5-7 Compiler Options
5-13 Link Editor Options

5-14 C Programming Notes
5-14 Static Strings
5-14 String Constants
5-15 Function Prototyping
5-16 Automatic Register Allocation
5-16 Declarations of External Variables and Functions
5-17 typedef and Type Modifiers

5-17 Legal Considerations
5-17 Distribution
5-18 GNU CC General Public License
5-19 Copying Policies
5-20 No Warranty

Chapter 5
The GNU C Compiler

The C compiler used on NeXT 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 NeXT 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.

This chapter Copyright © 1988 by Free Software Foundation, Inc. and Copyright © 1990 by NeXT Computer, Inc.

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.

For references to C documentation that adheres to the ANSI C standard, see "Suggested Reading" in the *NeXT Technical Summaries* manual of the *NeXT Developer's Library.*

GNU CC Command Options

The GNU C compiler is accessed with the **cc** command (note that on most UNIX systems **cc** would access the UNIX C compiler). This command accepts options and file names as operands. Multiple single-letter options may *not* be grouped: **-dr** is very different from **-d -r**.

When you invoke **cc** (the GNU compiler driver), it normally performs the following operations:

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

Files whose names end in ".c" are taken as C source files to be preprocessed and compiled; compiler output files plus any input files with names ending in ".s" are preprocessed and assembled; then the resulting object files, plus any other input files, are linked together to produce an executable file.

Command options allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker, in which case the output consists of object files produced by the assembler.

Certain command options are passed to one stage of the compilation process. For example, some options control just the preprocessor (**cpp**) and others just the compiler proper (**cc1**). Other options control the assembler and linker.

Global Compilation Options

The options that control the overall compilation process are listed below. This list includes the options that determine whether to link and whether to assemble.

-o *file* Place output in file *file*. This applies to whatever type of output is being produced; it could be an executable file, an object file, an assembler file, or preprocessed C code.

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

-c Compile or assemble the source files, but don't link. Produce object files with names made by replacing ".c" or ".s" with ".o" at the end of the input file names. Do nothing at all to object files specified as input.

-S Compile into assembler code but don't assemble. The assembler output file name is made by replacing ".c" with ".s" at the end of the input file name. Do nothing at all to assembler source files or object files specified as input.

-E Run only the C preprocessor. Preprocess the C source files and direct the results to the standard output.

-v Compile verbosely. The compiler driver program displays the commands it executes as it runs the preprocessor, compiler proper, assembler, and linker. Some of these are directed to print their own version numbers.

-vt Show timing information for each of the passes run by the **cc** command.

-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.

-nostdlib Don't use the standard system libraries and startup files (**-lcrt0.o** and **-lc**) when linking. Only the files you specify will be passed to the linker.

C Preprocessor Options

These C compiler options control the C preprocessor, which is run on each C source file and assembler file before actual compilation. If you use the **-E** option, nothing is done except C preprocessing. Some of these options make sense only together with **-E** because they request preprocessor output that isn't suitable for actual compilation.

-C Tell the preprocessor not to discard comments. Used with the **-E** option.

-I*dir* Search the directory *dir* for header 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 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 have specified with **-I** options (and the current directory, if appropriate) are searched.

Between **-nostdinc** and **-I-**, you can eliminate all directories from the search path except those you specify.

-M Tell the preprocessor to produce a rule suitable for **make** describing the dependencies of each source file. For each source file, the preprocessor produces one **make** rule whose target is the object file name for that source file and whose dependencies are all the files **#include**d in it. This rule may be a single line or may be continued with backslash-newline if it's long.

-M implies **-E** (that is, run only the C preprocessor).

-MD *file* This is similar to **-M**, but it writes the dependency information to *file* at the same time that the regular preprocessor output is directed to the standard output.

-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.

-MM implies **-E** (that is, run only the C preprocessor).

-MMD *file*
This is similar to **-MM**, but it uses the Mach-style **make**-depend switch, which writes dependency information to *file* at the same time that the regular preprocessor output is directed to the standard output.

-D*macro* Define macro *macro* with the empty string as its definition.

-D*macro=definition*
Predefine *macro* as a macro, with definition *definition*.

-U*macro* Undefine macro *macro*.

-T Support ANSI C trigraphs (the **-ansi** option also has this effect). Trigraphs are three-character sequences, all starting with **??**, that are defined by ANSI C to stand for single characters (these sequences allow users to use the full range of C characters, even if their keyboards don't implement the full C character set). For example, **??/** stands for \ so **??/n** is a character constant for newline.

-traditional
Attempt to support some aspects of traditional C preprocessors. Specifically:

- Comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

- Single and double quotation marks are ignored when scanning macro definitions, so that macro arguments can be replaced even within a string or character constant. Quotation marks are also ignored when skipping text inside a failing conditional directive.

-Wcomments
Warn whenever a comment-start sequence /* appears in a comment.

-Wtrigraphs
Warn whenever ANSI trigraphs are used.

Compiler Options

The following options control the details of C compilation (that is, just the portion of the compilation process related to **cc1**, the compiler proper).

-ansi Support all ANSI-standard C programs. This turns off certain features of GNU C that are incompatible with ANSI C, and enables the infrequently used ANSI trigraph feature.

The **-ansi** option doesn't cause non-ANSI programs to be rejected gratuitously. For that, **-pedantic** is required in addition to **-ansi**.

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.

-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.

-traditional

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

- All **extern** declarations take effect globally even if they're written inside a function definition. This includes implicit declarations of functions.
- The keywords **typeof**, **inline**, **signed**, **const**, and **volatile** aren't recognized.
- 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.

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

-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. With **-O**, the compiler tries to reduce code size and execution time. Some of the **-f** options described below turn specific kinds of optimization on or off.

-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 didn't 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, this makes it possible to debug optimized output if necessary.

-w Inhibit all warning messages.

-W Display extra warning messages if automatic variables are used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that's computed only when optimizing. They occur only for variables that are candidates for register allocation. Therefore, they don't occur for a variable that's 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're in registers.

There may be no warning about a variable that's used only to compute a value that itself is never used, because such computations may be deleted by the flow analysis pass before the warnings are displayed.

These warnings are made optional because GNU CC isn't smart enough to see that the code may actually be correct even though it appears 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.

- 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 can't 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 when there's no problem, because **longjmp()** can't be called at the place that 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 inspire 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.

-Wimplicit
Warn whenever a function 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, and whenever a function is declared static but never defined.

-Wall All the above **-W** options combined.

-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 ***.

-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. At compile time these warnings will help you find 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, setting this option will just be a nuisance; this is why **-Wall** doesn't request these warnings.

-pg Generate extra code to write profile information suitable for the analysis program **gprof**.

-d*options* Make debugging dumps at times specified by *options*. Here are the possible options:

t	Dump parse tree (*file*.**tree**)
r	Dump after RTL generation (*file*.**rtl**)
j	Dump after first jump optimization (*file*.**jump**)
J	Dump after last jump optimization (*file*.**jump2**)
s	Dump after CSE (*file*.**cse**)
L	Dump after loop optimization (*file*.**loop**)
f	Dump after flow analysis (*file*.**flow**)
c	Dump after instruction combination (*file*.**combine**)
l	Dump after local register allocation (*file*.**lreg**)
g	Dump after global register allocation (*file*.**greg**)
m	Display statistics on memory usage, at the end of the run

-pedantic Issue all the warnings demanded by strict ANSI-standard C; reject all programs that use forbidden extensions.

Valid ANSI-standard 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're rejected.

-f[**no-**]*flag* Specify machine-independent flags. Most flags have both positive and negative forms. For example, the negative form of **-ffoo** would be **-fno-foo**. In the list below, only one of the forms is shown—the one that's not the default. You can derive the other form by either removing or adding **no-** after the initial "f".

-fpcc-struct-return

Use the same convention for returning **struct** and **union** values that's used by PCC-compiled code. This convention is less efficient for small structures, and on many machines it fails to be reentrant; but it has the advantage of allowing intercallability between GCC-compiled code and PCC-compiled code.

-ffloat-store
Don't store floating-point variables in registers. This prevents undesirable excess precision due to the floating registers keeping more precision than a **double** is supposed to have.

For most programs, the excess precision does no harm, but a few programs rely on the precise definition of IEEE floating point. Use **-ffloat-store** for such programs.

-fno-asm
Don't recognize **asm**, **inline**, or **typeof** as a keyword. These words may then be used as identifiers.

-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.

-fstrength-reduce
Perform the optimizations of loop strength reduction and elimination of iteration variables.

-fcombine-regs
Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to **-O**.

-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.

-finline-functions
Integrate all simple functions into their callers. The compiler decides which functions are simple enough to be worth integrating.

If all calls to a given function are integrated, and the function is declared **static**, then the function normally isn't produced as assembler code in its own right.

-fkeep-inline-functions
Produce a separate run-time callable version of the function. Do so even if all calls to the function are integrated and the function is declared **static**.

-fwritable-strings
Store string constants in the writable data segment and don't make them unique. This is for compatibility with old programs that assume they can write into string constants. Writing into string constants is a very bad idea; "constants" should be constant.

-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.

-fvolatile
Consider all memory references through pointers to be volatile.

-funsigned-char
Let the type **char** default to unsigned, like **unsigned char**, rather than signed, like **signed char**.

-fsigned-char
Let the type **char** default to signed, like **signed char**.

-ffixed-*reg*
Treat the register *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer or 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.

-fcall-used-*reg*
Treat the register *reg* as an allocatable register that's 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.

-fcall-saved-*reg*
Treat the register *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.

Never use 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, or in a register in which function values may be returned.

Link Editor Options

These options control the **ld** link editor, which has been modified to support Mach-O files and shared libraries. See Chapter 8, "Mach Object Files," for more information about Mach-O files and Chapter 1, "Putting Together an Application," for more information about shared libraries. The UNIX manual page for **ld**(1) describes many other options recognized by the **ld** command.

-l*library* Search a standard list of directories for a library named *library*, which is actually a file named **lib*library*.a**. The linker uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with **-L**.

Normally the files found this way are library files—archive files whose members are object files. The linker handles an archive file by scanning through it for members that define symbols that have so far been referenced but not defined. But if the file that's found is an ordinary object file, it's linked in the usual fashion. The only difference between using an **-l** option and specifying a file name is that **-l** searches several directories.

-L*dir* Add directory *dir* to the list of directories to be searched by **-l**.

-sectcreate *segment section file*
Create a section named *segment* and a section named *section* in that segment containing the contents of *file*. The known segments **__TEXT** and **__DATA** shouldn't be used as the segment name.

-i*definition*:*indirect*
Create an indirect symbol for the symbol name *definition* which is defined to be the same as the symbol name *indirect* (which is taken to be undefined). When a definition of the symbol named *indirect* is linked, both symbols take on the defined type and value.

-Z Inhibit the searching of the default directories for **-lx** arguments.

If you add segments to Mach-O files with the **-segcreate** flag, the contents of *file-name* go into the segment (the **cc** command also understands this set of flags). This will also work with **atom**, the "a.out to Mach-O" converter. These segments are mapped into the address space of the executable file, and the contents can be read (and written) by the executable file. Note that if you write it, it doesn't go back into the executable file. It's just like initialized data (copy-on-write). It's intended to be used for things such as the icons and the archive. To get to these segments from your code, use:

```
const struct load_command *
getsegbyname(char *name);
```

You pass the name of the segment for which you want the pointer to the segment structure, and it will return the pointer (or 0 if the pointer doesn't exist). The structure is read-only.

C Programming Notes

This section contains miscellaneous notes about programming in C on a NeXT computer. It also describes some incompatibilities between GNU C and traditional non-ANSI versions of C.

Static Strings

Initialized strings are normally put in the text segment by the GNU compiler, and attempts to write to them cause segmentation faults. However, some programs depend 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";
```

String Constants

GNU CC normally makes string constants read-only. 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.

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" in the *NeXT Technical Summaries* manual of the *NeXT Developer's Library* 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.

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 "GNU CC 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 sections entitled "GNU CC 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 can obtain a complete machine-readable copy of any NeXT-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) by writing 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 NeXT 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.

GNU CC 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 GNU CC. 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 GNU CC, that you receive source code or else can get it if you want it, that you can change GNU CC 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 GNU CC, 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 GNU CC. If GNU CC 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 GNU CC.

Copying Policies

1. You may copy and distribute verbatim copies of GNU CC source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy a valid copyright notice "Copyright © 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 GNU CC 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 GNU CC 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 GNU CC 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 GNU CC (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 GNU CC except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GNU CC 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.

5. If you wish to incorporate parts of GNU CC into other free programs whose distribution conditions are different, write to the Free Software Foundation at 675 Mass. Ave., Cambridge, MA 02139. We have not yet worked out a simple rule that can be stated here, but we will often permit this. We 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.

Your comments and suggestions about our licensing policies and our software are welcome! Please contact the Free Software Foundation, Inc., 675 Mass. Ave., Cambridge, MA 02139, or call (617)876-3296.

No Warranty

BECAUSE GNU CC IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE GNU CC “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 GNU CC IS WITH YOU. SHOULD GNU CC PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL RICHARD M. STALLMAN, THE FREE SOFTWARE FOUNDATION, INC., AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE GNU CC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) GNU CC, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Chapter 6
The GNU C Preprocessor

6-4 Global Transformations

6-4 Preprocessor Commands

6-5 Header Files
6-5 Uses of Header Files
6-5 The **#include** Command

6-6 Macros
6-6 Simple Macros
6-8 Macros that Take Arguments
6-10 Predefined Macros
6-12 Nonstandard Predefined Macros
6-12 Stringification
6-13 Concatenation
6-14 Undefining Macros
6-15 Redefining Macros
6-15 Pitfalls and Subtleties of Macros
6-16 Improperly Nested Constructs
6-16 Unintended Grouping of Arithmetic
6-17 Swallowing the Semicolon
6-18 Duplication of Side Effects
6-19 Self-Referential Macros
6-20 Separate Expansion of Macro Arguments
6-21 Cascaded Use of Macros
6-22 Inability to Define a Macro that Produces a # Character
6-22 Macro Arguments inside String Constants

6-22 Conditionals
6-23 Syntax of Conditionals
6-23 The **#if** Command
6-24 The **#else** Command
6-24 The **#elif** Command
6-25 Keeping Deleted Code for Future Reference
6-25 Conditionals and Macros
6-26 The **#error** Command

6-27 Pragmas

6-27 Combining Source Files

6-28 C Preprocessor Output

6-28 Invoking the C Preprocessor

Chapter 6
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 must use the options **-T**, **-undef**, and **-pedantic**. See the section "Invoking the C Preprocessor."

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").

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 # 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 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.

#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.)

#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.

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.

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.

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.

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.

__LINE__

This macro expands to the current input line number, in the form of a decimal integer constant.

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__);
```

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 11 characters and looks like "Jan 29 1987".

__TIME__

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like "23:59:01".

__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 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 by NeXT:

__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 NeXT 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 (including the NeXT Computer).

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.

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 definition 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.

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.

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 **char**s.) 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 white-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.

The best solution to this problem 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 two special cases:

- Nested invocations of a macro
- Macros that invoke other macros that stringify or concatenate

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.

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. 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**.

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.

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. 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 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 Command

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.

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
```

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.

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."

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**. 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.

-T 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 *file* This is similar to **-M**, but it writes the dependency information to *file* at the same time that the regular preprocessor output is directed to the standard output.

-MM Like **-M** but mention only the files included with **#include "*file*"**. System header files included with **#include** <*file*> are omitted.

-MMD*file* This is similar to **-MM**, but it uses the Mach-style **make**-depend switch, which writes dependency information to *file* at the same time that the regular preprocessor output is directed to the standard output.

-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.

Chapter 7
The GNU Source-Level Debugger

7-5 Summary of GDB

7-6 Compiling Your Program for Debugging

7-7 Running GDB
7-8 Specifying Files to Debug
7-8 Specifying GDB Modes
7-9 Editing GDB Commands
7-9 Viewing the Command History
7-9 Editor Commands
7-12 Running GDB in a GNU Emacs Buffer

7-13 Startup Files

7-13 GDB Commands for Specifying Files

7-14 Running Your Program under GDB
7-15 Your Program's Arguments
7-15 Your Program's Environment
7-16 Your Program's Working Directory
7-16 Your Program's Input and Output
7-17 Debugging an Already Running Process

7-17 Stopping and Continuing
7-18 Signals
7-19 Breakpoints
7-20 Setting Breakpoints
7-21 Clearing Breakpoints
7-21 Disabling Breakpoints
7-22 Break Conditions
7-24 Executing Commands at a Breakpoint
7-25 Continuing
7-26 Stepping

7-27 Examining the Stack
7-27 Stack Frames
7-28 Backtraces
7-28 Selecting a Frame
7-29 Information about a Frame

7-29 Examining Source Files
7-30 Printing Source Lines
7-31 Searching Source Files
7-32 Specifying Source Directories

7-33 Examining Data
7-33 Expressions
7-34 Program Variables
7-34 Artificial Arrays
7-35 Output Formats
7-36 Examining Memory
7-38 Automatic Display
7-38 Value History
7-39 Convenience Variables
7-40 Registers

7-41 Examining the Symbol Table

7-43 Setting Format Options

7-43 Debugging PostScript

7-44 Debugging Objective-C
7-44 Method Names in Commands
7-45 Command Descriptions
7-45 The **info** Command
7-45 The **pclass** Command
7-46 The **print** Command
7-47 The **set** Command
7-47 The **step** Command

7-48 Debugging Mach Threads

7-48 Debugging NeXT Core Files

7-49 Altering Execution
7-49 Assignment to Variables
7-49 Continuing at a Different Address
7-50 Returning from a Function

7-50 Defining and Executing Sequences of Commands
7-50 User-Defined Commands
7-51 Command Files
7-52 Commands for Controlled Output

7-52 Legal Considerations
7-53 Distribution
7-54 GDB General Public License
7-54 Copying Policies
7-56 No Warranty

Chapter 7
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 by NeXT 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 NeXT-specific extensions to GDB. These NeXT extensions provide full compatibility with standard GDB, while offering the following additional features useful for developing programs within the NeXT 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.

This chapter Copyright © 1988 by Free Software Foundation, Inc. and Copyright © 1990 by NeXT Computer, Inc.

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 NeXT computer, you'll normally 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

A NeXT-extended library routine adds Emacs-style command-line editing to the standard GDB command-line interface (this mode has some superficial differences from standard Emacs commands). You select the Emacs editing mode with the following command:

```
editmode emacs
```

Alternatively, you can put this command in your home directory in a file named **.gdbinit**. During startup GDB reads and executes the commands in this file (see "Startup Files" below). In this way, you can ensure that GDB is in the proper editing mode whenever it starts.

Viewing the Command History

In the Emacs editing mode, the **history** command can be used to display a list of all commands executed during the current session. This is useful because, unlike the standard GDB mode, the Emacs editing mode 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. By typing Control-P repeatedly, you can step back through each of the commands that were issued since the beginning of the session. The command Control-N steps forward through the history buffer.

Editor Commands

The following list of Emacs-mode 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 (Backspace)
Control-F	Move forward one character (ForwardChar)
Esc b	Move back one word (BackwardWord)
Esc f	Move forward one word (ForwardWord)
Control-A	Move to beginning of line (BeginningOfLine)
Control-E	Move to end of line (EndOfLine)

Deletion and Restoration Commands

Control-D	Delete current character (DeleteCurrentChar)
Delete or Control-H	Delete previous character (DeletePreviousChar)
Esc d	Delete current word (DeleteWord)
Esc h	Delete previous word (EraseWord)
Control-K	Kill forward to end of line (KillToEOL)
Control-W	Kill region (KillRegion)
Control-Y	Restore previous kill from buffer (YankKillBuffer)

Search Commands

Esc /	Search forward (IncrementalSearchForward)
Esc ?	Search backward (IncrementalSearchReverse)
Esc	Exit search mode

Macro Commands

Control-X (	Begin keyboard macro definition (StartRemembering)
Control-X)	End keyboard macro definition (StopRemembering)
Esc n	Define named macro (DefineNamedMacro)
Esc x	Execute named macro (ExecuteNamedMacro)
Esc e	Execute unnamed macro (ExecuteUnnamedMacro)
Control-X Control-R	Load named macro file (LoadMacroFile)
Control-X Control-S	Save macro file (SaveMacroFile)

History Commands

Esc <	Move to beginning of history file (BeginningOfHistory)
Esc >	Move to end of history file (EndOfHistory)
Control-N	Go to next history file entry (NextHistEntry)
Control-P	Go to previous history file entry (PreviousHistEntry)
Esc s	Search history file forward (IncrementalSearchHistoryForward)
Esc r	Search history file backward (IncrementalSearchHistoryReverse)

Miscellaneous Commands

Control-L	Clear screen (ClearScreen)
Control-R	Redisplay current command line (Redisplay)
Control-Q	Insert a literal character (InsertLiteralChar)
Control-I	Insert a Tab (Tab)
Control-T	Transpose characters (TransposeChars)
Control-U *n*	Repeat following command *n* times (Repetition)
Control-Z	Suspend debugger, return to shell (Suspend)
Control-@	Set mark (SetMark)

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)).

You can associate (or "bind") a command with a different key combination by placing a definition in a file named **.bindings**. When you give the GDB command **editmode emacs**, GDB reads **.bindings** files in both the home directory and the current directory. A **.bindings** file can contain any GDB commands, but its intended use is to change the default key bindings. The following commands illustrate the syntax for commands in a **.bindings** file:

```
bind-to-key BeginningOfHistory "\eh"
bind-to-key SaveMacroFile "\^X\^W"
```

In these examples, the Esc character is represented as **\e** and Control characters are represented as **\^** followed by the appropriate letter (for example, Control-X is represented as **\^X**).

Each command description earlier in this section includes in parentheses the command name as it should appear in the **.bindings** file.

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 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.

exec-file *file*

Specify that the program to be run is found in *file*. If you don't specify a directory and the file isn't found in GDB's working directory, GDB will use the environment variable PATH as a list of directories to search, just as the shell does when looking for a program to run.

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.

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.

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.

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.

While all three 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.

A variant of the **run** command is **really-run**. This command is the same as **run**, but it doesn't ask for confirmation.

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.

info environment
Print the names and values of all the environment variables that are given to your program when it's started. This command can be abbreviated as **i env**.

info environment *varname*
Print the value of the environment variable *varname*.

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. This command can be abbreviated as **set e**.

delete environment *varname*

Remove 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 command can be abbreviated as **delete e**.

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 GDB's working directory to *dir*.

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 NeXT 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. (The usual way to find out the process ID of the process is with the **ps** utility.)

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. 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**.

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.

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 signal
: 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.

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:

disable *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.

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).

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).

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.

cont *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 **cont** 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 **cont** (continue) command:

cont 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, **cont** 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 **cont** 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). 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**.

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.

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. If you specify an argument, the command stops after *n* frames. You can abbreviate this command as **bt**.

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; all of them finish by printing a brief description of the stack frame just selected.

frame *n*
: Select 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 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 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.

down *n*
: Select 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.

All these commands 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.

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 *linenum*
Print the starting and ending addresses of the compiled code for source line *linenum*.

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.

To see the current source path, use the **info directories** command:

info directories
Print the source path, showing which directories it contains.

When you start GDB, its source path contains just the current working directory. To add other directories, use the **directory** command or the **idir** command:

directory *dirname*
Add directory with the pathname *dirname* to the end of the source path.

directory Reset the source path to just the current working directory of GDB. This requires confirmation.

The **directory** command adds directories to the end of the source path, so it isn't useful if you want to add a directory to the search path and have it be searched before other directories. In this case, you should use the **idir** command. The **idir** command is similar to the **directory** command, but it inserts a directory at the front of the search path rather than at the end.

idir *dirname*
The **idir** command inserts the directory with the pathname *dirname* at the front of the search path, causing that directory to be searched first. With no argument, **idir** resets the search path to GDB's current working directory.

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). *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.

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.

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.

info history [*n*]
With no argument, print the last ten values in the value history with their item numbers. With an argument, print ten history values centered on history item *n*. **info history** doesn't change the history.

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.

info convenience
: Print a list of convenience variables used so far, and their values. You can abbreviate this command as **i con**.

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.

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*.

Setting Format Options

The **format** command is a NeXT extension to GDB. This command lets you set options that affect the format of GDB output.

format If no arguments are specified, this command prints the current format options.

format maximum-string *n*
Set the maximum number of characters printed in a character string to the decimal number *n*. The default value is 200.

format struct-indent *n*
If *n* is equal to 1 (the default), structures are indented when printed, with a line break after every field in the structure. If *n* is equal to 0, structure fields are printed contiguously with commas separating each field.

format union-print *n*
If *n* is equal to 1, all fields of a union are printed when the union is printed. If *n* is equal to 0 (the default), only the names of the variants are printed. You must specify the variants of the union in which you're interested in order to see it printed in its entirety.

format maximum-line *n*
The value of *n* indicates the maximum number of lines to be printed before pausing the display of output. This should be the number of lines on the screen minus a few (depending on your preference). If the value of *n* is 0, the output never pauses (useful if GDB is running in a scrollable Shell window).

Debugging PostScript

This section describes three commands that are useful when debugging PostScript source files.

These commands aren't built-in commands; rather, NeXT 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

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 by NeXT, and some are previously existing GDB commands that have been extended by NeXT.

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.

info syms Print all object files that contain symbol table information. They're divided into two groups, based on whether or not their symbols have been read.

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 pclass Command

The **pclass** (print class) command displays instance and class methods for the class *classname*:

pclass *classname*
Display instance and class methods for the class *classname*. For a listing of all methods in your application, use the **info selectors** command described above.

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 by NeXT 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 NeXT Core Files

NeXT has extended GDB to allow debugging of NeXT core files, which are in the Mach-O file format. These files are very large, so they aren't generated by default. In order for core files to be generated, you must raise your core file limit by typing one of the following commands at the UNIX prompt:

```
unlimit core
limit core 5m
```

The first command allows core files of any size to be created. The second command only allows core files less than 5 megabytes to be created. The maximum core file size is up to you. Note that because of the way core files are created under Mach, even a small application can create a core file of several megabytes. 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 "Formats" 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).

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 NeXT-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 NeXT 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.

5. If you wish to incorporate parts of GDB into other free programs whose distribution conditions are different, write to the Free Software Foundation at 675 Mass. Ave., Cambridge, MA 02139. We have not yet worked out a simple rule that can be stated here, but we will often permit this. We 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.

Your comments and suggestions about our licensing policies and our software are welcome! Please contact the Free Software Foundation, Inc., 675 Mass. Ave., Cambridge, MA 02139, or call (617)876-3296.

No Warranty

BECAUSE GDB IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE GDB "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 GDB IS WITH YOU. SHOULD GDB PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL RICHARD M. STALLMAN, THE FREE SOFTWARE FOUNDATION, INC., AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE GDB AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) GDB, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Chapter 8
Mach Object Files

8-4 The Mach Header

8-5 The Load Commands
8-6 The LC_SEGMENT Load Command
8-9 The LC_SYMTAB Load Command
8-11 The LC_THREAD and LC_UNIXTHREAD Load Commands
8-11 The LC_LOADFVMLIB and LC_IDFVMLIB Commands

8-12 Relocation Information

8-13 The Makeup of Executable Object Files

Chapter 8
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 NeXT computers.

The current Mach object format is still evolving at Carnegie Mellon. NeXT has made enhancements that 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 **sys/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) */
```

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 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 that of a UNIX 4.3BSD **a.out** file: The string table offset and size are explicitly defined, and the symbol table and string table themselves are located at the end of the file rather than 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 */
};
```

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
-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

Index

preprocessor directive 6-13
$ convenience variable indicator 7-39
$_ convenience variable 7-40
$__ convenience variable 7-40
@ binary operator 7-34

Align Left command in Edit 3-33
Align Right command in Edit 3-33
alignment in Edit
  paragraph 3-33
  tab stop 3-8
AppInspector application 4-3
  commands 4-13
application
  debugging 1-18, 4-3, 7-5
  development process 1-3
  directory 1-28
  document extension 1-29
  installation 1-27
  project 1-6
  search path 1-27
Application menu
  in AppInspector 4-13
  in MallocDebug 4-19
args GDB command 7-15
artificial arrays in GDB 7-34
attach GDB command 7-17
automatic register allocation 5-16

backtrace 1-25, 7-28
backtrace GDB command 7-28
.bindings file 7-11
break GDB command 7-20
breakpoints in GDB
  clearing 7-21
  conditional 7-22
  continuing program execution 7-25
  disabling 7-21
  executing commands at 7-24
  setting 7-20
Browse menu in AppInspector 4-14

C compiler 1-19, 5-3
  compiler-specific options 5-7
  compiling your program for debugging 7-6
  dumps 5-10
  examples 1-20
  general public license 5-18
  global compilation options 5-4
  legal considerations 5-17
  link editor options 5-13
  optimization 5-7
  preprocessor options 5-5
  warnings 1-19
C preprocessor 6-3
  commands 6-4
  conditionals 6-22
  global transformations 6-4
  invoking 6-28
  options to C compiler 5-5
  output 6-28
C programming notes 5-14
  automatic register allocation 5-16
  external declarations 5-16
  function prototypes 5-15
  static strings 5-14
  string constants 5-14
  typedef and type modifiers 5-17
call stack *See* stack
cc shell command 5-3
cd GDB command 7-16
Center command in Edit 3-33
Check Spelling command in Edit 3-27
class definition files 1-7
classes
  browsing 4-7
  finding 4-11
  inspecting 4-7, 4-10
Classes command in AppInspector 4-7
Clear Buffer command in Terminal 2-12
clear GDB command 7-21
Clear Monitors command in ProcessMonitor 4-25
Close command in MallocDebug 4-19
Command command in Edit 3-18, 3-35
commands GDB command 7-24
compiler *See* C compiler
condition GDB command 7-23

conditional, C preprocessor
with macro 6-25, 6-22
syntax 6-23
cont GDB command 7-23, 7-25
Contract All command in Edit 3-34
Contract Sel command in Edit 3-34
convenience variables in GDB 7-39
Copy command in Terminal 2-12
Copy PS command in Edit 3-23
Copy Ruler command in Edit 3-33
copying policy 5-19
core-file GDB command 7-14
core files
debugging 7-48
specifying in GDB 7-14
custom libraries 1-12
Cut command in Terminal 2-12

damaged nodes 4-18
debugging
an already running process 7-17
an application 1-18, 4-3, 7-5
core files 7-48
an executable file 1-20, 1-23
Mach threads 7-48
Objective-C 7-44
PostScript 7-43
define GDB command 7-51
#define preprocessor directive 6-6, 6-8
delete environment GDB command 7-16
delete GDB command 7-21
detach GDB command 7-17
developer applications 4-3
directory GDB command 7-32
disable GDB command 7-22
display GDB command 7-38
Display PostScript
Inspector in ProcessMonitor 4-23
document GDB command 7-51
down GDB command 7-28

echo GDB command 7-52
Edit application 3-3
command-line options 3-3
commands 3-20
and UNIX 3-18
windows 3-5
Edit menu
in Edit 3-23
in Terminal 2-12
editmode GDB command 7-9
#elif preprocessor directive 6-24
#else preprocessor directive 6-24
Emacs
commands in Edit 3-17
GDB interface 7-12
mode in GDB 7-9
enable GDB command 7-22
end GDB command 7-24, 7-51
environment variables 7-15
#error preprocessor directive 6-26
exec-file GDB command 7-13, 7-19
executable object file 8-13
Expand All command in Edit 3-34
Expand Sel command in Edit 3-34
expressions in GDB 7-33
extension 1-29
external declarations and the C compiler 5-16

file extension 1-29
File menu in Edit 3-20
file package 1-28
Find command in AppInspector 4-11
Find menu
in Edit 3-28
in Terminal 2-14
Find Panel command
in Edit 3-28
in Terminal 2-14
finish GDB command 7-26
flush
GDB command 7-43
PostScript operator 4-29
Font menu in Edit 3-32
Font Panel command in Terminal 2-14
format GDB command 7-43
Format menu
in Edit 3-31
in Terminal 2-13
format output in GDB 7-43
forward-search GDB command 7-31
frame GDB command 7-28, 7-29
function prototypes 5-15

garbage detection 4-18

GDB 7-5
  breakpoints *See* breakpoints in GDB
  convenience variables 7-39
  customizing 7-13, 7-50
  data display 7-33, 7-38
  Emacs editing mode 7-9
  files to debug, specifying 7-8, 7-13
  legal considerations 7-52
  memory, examining 7-36
  output format 7-35, 7-43
  program execution 7-14, 7-25, 7-49
  registers 7-40
  signals 7-18
  source files 7-29, 7-31, 7-32
  stack *See* stack *and* stack frame
  stepping 7-26
  tracing techniques 1-25
  value history 7-38
  variable assignment 7-49
  See also debugging
gdb shell command 7-7
.gdbinit file 7-13
  preventing execution of 7-8
GNU C preprocessor *See* C preprocessor
GNU CC *See* C compiler
GNU debugger *See* GDB
GNU Emacs *See* Emacs

handle GDB command 7-18
hanging indent in Edit 3-8
header files 6-5
help GDB command 7-51
Hide Non Apps command in ProcessMonitor 4-26
Hide Ruler command in Edit 3-33
history GDB command 7-9

icon header file 1-9
idir GDB command 7-32
#if preprocessor directive 6-23
ignore GDB command 7-23
image files 1-10
#import preprocessor directive 6-5
#include preprocessor directive 6-5
indentation in Edit 3-8, 3-14
info address GDB command 7-42
info args GDB command 7-29
info breakpoints GDB command 7-19
info classes GDB command 7-45
info convenience GDB command 7-40
info directories GDB command 7-32
info display GDB command 7-38
info environment GDB command 7-15
info files GDB command 7-14, 7-48
info frame GDB command 7-29
info functions GDB command 7-42
info history GDB command 7-39
info line GDB command 7-31
  extended for Objective-C 7-45
info locals GDB command 7-29
info registers GDB command 7-41
info selectors GDB command 7-45
info signal GDB command 7-18
info sources GDB command 7-42
info syms GDB command 7-45
info types GDB command 7-42
  extended for Objective-C 7-45
info variables GDB command 7-42
Inspect command in AppInspector 4-7
instances
  browsing 4-5
  inspecting 4-7
Interface Builder 1-3
  makefiles 1-13
  tasks in development process 1-5
interface files 1-9

jump GDB command 7-49
Jump to Selection command in Terminal 2-15

kill GDB command 7-14

LC_IDFVMLIB load command 8-11
LC_LOADFVMLIB load command 8-11
LC_SEGMENT load command 8-6
LC_SYMTAB load command 8-9
LC_THREAD load command 8-11
LC_UNIXTHREAD load command 8-11
ld shell command *See* link editor
left margin
  in Mail 3-7
libraries 1-11
  customizing 1-12
limit shell command 7-48
Line Range command in Edit 3-30
link editor 1-19
  examples 1-20
  options to C compiler 5-13
  warnings 1-19
list GDB command 7-30
load commands in object file
  LC_IDFVMLIB 8-11
  LC_LOADFVMLIB 8-11
  LC_SEGMENT 8-6
  LC_SYMTAB 8-9
  LC_THREAD 8-11
  LC_UNIXTHREAD 8-11

Mach
  debugging threads 7-48
  Inspector in ProcessMonitor 4-22
  object file 8-3
Mach-O 8-3
macro, C preprocessor
  arguments *See* macro arguments
  cascaded use 6-21
  with conditional 6-25
  duplication of side effects 6-18
  expansion 6-6, 6-7, 6-8
  pitfalls and subtleties 6-15
  predefined 6-10, 6-12
  redefining 6-15
  self-referential 6-19
  simple 6-6
  stringification 6-12
  undefining 6-14
macro arguments, C preprocessor 6-8
  concatenation 6-13
  inside string constants 6-22
  separate expansion of 6-20
main file 1-7
make program 1-12
Make Rich Text command in Edit 3-33
makefile 1-12
  customizing 1-18
  postamble 1-17
  preamble 1-16
  running 1-18
Malloc
  debugger 4-15
  Inspector in ProcessMonitor 4-24
MallocDebug application 4-15
  commands 4-19
Manager menu in Edit 3-22
Manual command in Edit 3-36
margins in Edit 3-7, 3-8
Match command in Edit 3-24
memory
  examining in GDB 7-36
  finding leaks 4-18
  inspecting with AppInspector 4-9
  usage 4-18, 4-25
Monitor menu in ProcessMonitor 4-26

Nest command in Edit 3-24
New command in Terminal 2-11
next GDB command 7-26
NeXT libraries 1-11
nexti GDB command 7-26

object file 8-3
  executable 8-13
  header 8-4
  relocation information 8-12
Objective-C
  debugging 7-44
Objects command in AppInspector 4-5
Open command
  in AppInspector 4-3, 4-4
  in MallocDebug 4-16
Open Directory command in Edit 3-22
Open Selection command in Edit 3-21
openfile shell command 3-4
optimization 5-7
orderwindow PostScript operator 4-29
otool shell command 8-15
output format in GDB 7-35, 7-43
output GDB command 7-52

Page Layout command
  in Edit 3-31
  in Terminal 2-14
Paste command in Terminal 2-12
Paste Ruler command in Edit 3-33
pclass GDB command 7-45
Peep in AppInspector 4-12
pft utility 4-27
Pipe command in Edit 3-18, 3-35
postamble file 1-17
PostScript
  debugging 7-43
  Window Server interface 4-27
  See also Display PostScript
#pragma preprocessor directive 6-27
preamble file 1-16
Preferences command in Edit 3-11
  C options 3-15
  global options 3-12
  temporary settings 3-13
  text options 3-14
Preferences command in Terminal 2-4
  emulation preferences 2-6
  shell preferences 2-8
  window preferences 2-7
preprocessor *See* C preprocessor
print GDB command 7-33
  extended for Objective-C 7-46
  output formats 7-35
  value history 7-38
printf GDB command 7-52
printsyms GDB command 7-42

process
  ID 4-22
  inspecting 4-21
  Mach memory usage 4-23
  monitoring 4-20
  selecting 4-20
Processes menu in ProcessMonitor 4-26
ProcessMonitor application 4-20
  commands 4-25
project
  directory 1-6
  file 1-11
  makefile 1-13
  management files 1-11
prototype, function 5-15
ptype GDB command 7-42
public domain software 5-17
pwd GDB command 7-16

quit GDB command 7-7

really-run GDB command 7-14
registers
  in GDB 7-40
relocation information in object file 8-12
return GDB command 7-50
reverse-search GDB command 7-31
Rich Text Format in Edit 3-13, 3-33
right margin
  in Mail 3-7
RTF *See* Rich Text Format in Edit
ruler in Edit 3-6
ruler in Mail 3-7
run GDB command 7-14
  redirecting input and output 7-16
run-time tracing 4-12

Save As command in Edit 3-21
Save command in Edit 3-21
search GDB command 7-31
set environment GDB command 7-16
set GDB command 7-33
  extended for Objective-C 7-47
set prompt GDB command 7-7
shared makefile 1-14
Shell menu in Terminal 2-11
Show Non Apps command in ProcessMonitor 4-20, 4-26
Show Ruler command in Edit 3-33
shownops GDB command 7-43
showps GDB command 7-43
signal GDB command 7-19
signals in GDB 7-18
silent GDB command 7-24
sound files 1-10
Source command in Edit 3-36
source files 1-6, 1-8
  combining 6-27
  examining in GDB 7-29
  searching in GDB 7-31
  specifying directories in GDB 7-32
source GDB command 7-51
Spelling command in Edit 3-26, 3-27
stack
  backtrace 7-28
  examining 7-27
  selecting a frame 7-28
stack frame 7-27
  information about 7-29
  returning from 7-50
  selecting 7-28
Start Monitor command in ProcessMonitor 4-25
startup files for GDB 7-13
static strings and the C compiler 5-14
Steal Keys command in Terminal 2-12
step GDB command 7-26
  extended for Objective-C 7-47
stepi GDB command 7-26
stepping in GDB 7-26
string constants and the C compiler 5-14
stringification and macros 6-12
Structure menu in Edit 3-9, 3-34
symbol-file GDB command 7-14
symbol table
  examining in GDB 7-41
  specifying in GDB 7-14

tabs in Edit 3-8
tags file 3-19, 3-36
tbreak GDB command 7-20
Templates command in Edit 3-24
Terminal application 2-3
  commands 2-9
Text menu in Edit 3-32
thread-list GDB command 7-48
thread-select GDB command 7-48
Tools menu in Appinspector 4-14
tresume GDB command 7-48
tsuspend GDB command 7-48
tty GDB command 7-16
typedef and type modifiers 5-17

#undef preprocessor directive 6-14
Undelete command in Edit 3-23
undisplay GDB command 7-38

UNIX
displaying manual pages in Edit 3-36
piping output into Edit 3-18
shell 2-4
using a tags file in Edit 3-19
utility commands in Edit 3-35
unlimit shell command 7-48
Unnest command in Edit 3-24
until GDB command 7-26
up GDB command 7-28
Update command in ProcessMonitor 4-20, 4-26
Update Directory command in Edit 3-22
User Commands menu in Edit 3-18, 3-35
User Pipes menu in Edit 3-18, 3-35
Utilities menu in Edit 3-34

value history in GDB 7-38
variables in GDB
altering values 7-49
convenience variables 7-39
environment variables 7-15
program variables 7-34
VT100 emulation 2-3

warnings 1-19
whatis GDB command 7-41
window PostScript operator 4-28
Window Server interface 4-27
windowdeviceround PostScript operator 4-29

x GDB command 7-36

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

Printed in U.S.A.
2912.00
12/90

Text printed on
recycled paper