ULTRIX

digital

Guide to Languages and Programming

Part Number: AA–ML94C–TE

ULTRIX

Guide to Languages and Programming

Order Number: AA-ML94C-TE
May 1991

Product Version: ULTRIX Version 4.2 or higher

digital equipment corporation
maynard, massachusetts

Contents

4 Checking Programs and Improving Performance

5 Library Routines and System Calls

Figures

Tables

About This Manual

This guide describes the tools and methods used to write application programs on the ULTRIX system. It gives an overview of the commands in the ULTRIX compiler system and describes the commands used to build, debug, and optimize programs. This guide also describes using C library routines, writing secure programs, and calling between the C and Pascal languages.

This guide does not list the syntax and definition of the elements of each language. The guide neither attempts to teach programmers how to write an application, nor does it attempt to teach the concepts of C, FORTRAN, Pascal or other languages.

Audience

The audience for this manual is the application programmer or system engineer who is already familiar with a programming language. This manual gives information about tools and concepts particular to programming on an ULTRIX system.

Organization

Chapter 1 describes the steps in application development and introduces the tools programmers use to develop applications.

Chapter 2 describes using the compilers and linkers provided with the ULTRIX system. This chapter also describes using the `make` utility to build programs.

Chapter 3 describes debugging your program using the `dbx` debugger.

Chapter 4 describes the profiling and optimization facilities that are available as part of the ULTRIX compiler system. You can use these facilities to increase the efficiency of your programs.

Chapter 5 gives an overview of the system calls and library routines and helps programmers decide which routines to use for a particular task. This chapter also describes using ULTRIX routines to read from and write to files and devices.

Chapter 6 describes controlling communication between processes running on an ULTRIX system. This chapter gives examples of using pipes, handling signals, and using sockets.

Chapter 7 provides security guidelines for designing and writing programs.

Chapter 8 describes the coding interfaces between C and Pascal and provides information for calling and passing arguments between those languages for both the RISC and VAX architecture.

Appendix A describes the extensions and modifications that are supported by the `cc` compiler that runs on the RISC architecture. The extensions and modifications are differences between `cc` on the RISC architecture and the C language defined by Kernighan and Ritchie.

Appendix B describes how the compiler groups C structures in storage for the RISC and VAX architectures.

Appendix C describes the issues involved in porting an application from the VAX architecture to the RISC architecture.

Appendix D describes how to debug the ULTRIX kernel `/vmunix` on the RISC and VAX architectures.

Related Documents

See the user's guides for the individual programming languages for descriptions of each language.

See the *Guide to Developing International Software* if you are writing programs for an international environment.

See the *Guide to Network Programming* if you are writing a network application.

See the *DECrpc Programming Guide* if you are developing an application based on DECrpc.

The *ULTRIX Reference Pages* contain reference information for the commands and tools that are described in this manual. The reference pages are available in printed form and online.

To view reference pages online, use the `man` or `whatis` commands.

The `-f` option to the `man` or `whatis` commands allows you to view a one-line summary of the specified topic name. Occasionally, the same topic name appears in more than one section of the *ULTRIX Reference Pages.* This situation occurs, for example, when a command and a system call have the same name. Section 1 of the reference pages describes using the command, while Section 2 describes using the system call. Using the `-f` option, you can determine where to look to read about each occurrence of a topic. For example, the following `man` command lists all occurrences of the `chmod` topic:

```
% man -f chmod
chmod (1)              - change file mode
chmod, fchmod (2)      - change mode of file
```

If you want to read only the information about the system call, specify Section 2 of the reference pages on the `man` command line. For example, the following command displays the `chmod`(2) reference page for the chmod system call and not its Section 1 (command) counterpart:

```
% man 2 chmod
```

To read general information about a group of commands or routines, display the `intro` reference page for a particular section. For example, to read information about the math library, display the `intro`(3m) reference page by entering the following command:

```
% man 3m intro
```

Conventions

`%`	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (`%`) is used to represent this prompt.
#	A number sign is the default superuser prompt.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in interactive examples to indicate system output. It is also used in code examples and other screen displays. In text, this typeface indicates the exact name of a command, option, partition, pathname, directory, or file.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
`cat`(1)	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat`(1) indicates that you can find the material on the `cat` command in Section 1 of the reference pages.
RISC Specific	Some information in this manual is specific to systems that run on the RISC architecture. This information is labeled RISC specific throughout the manual.
VAX Specific	Some information in this manual is specific to systems that run on the VAX architecture. This information is labeled VAX specific throughout the manual.

Introduction 1

This manual contains information about how to create and maintain programs and applications on an ULTRIX system. Use this manual in conjunction with the manuals in the documentation set for your programming language and with other manuals in the ULTRIX documentation set that contain detailed information about topics only summarized here.

The following topics are covered in this manual:

- Compiling, linking, running, and building programs
- Library routines and system calls
- Programming language interfaces
- Using object files and libraries
- Debugging tools
- Checking programs and improving performance
- Interprocess communication
- Messages and error handling
- Security guidelines

In some cases, all the relevant information you need is included in this manual. In other cases, a section contains overview information and then refers to other manuals that describe the topic in detail.

This manual applies to ULTRIX operating systems running on both the RISC and VAX platforms. You can assume that the environments behave in the same way unless differences are identified in the manual.

This chapter describes the phases you go through in developing an application and tells you which ULTRIX tools to use during those phases. The chapter also discusses the following topics:

- Specifications and design considerations
- Major software development tools
- Source file control
- Program installation tools

1.1 Application Development Phases

There are five major phases in developing a new application:

- Requirements and specifications
- Design
- Implementation
- Testing
- Installation and maintenance

The first phase involves outlining the requirements and specifications for the application. You need to answer the following types of questions:

- What tasks will the application perform?
- In what kind of environment will the application run?
- Who will be using the application?
- Does the application need to be portable?

For example, you need to know whether users will have workstations or window terminals or whether they will be working on character-cell terminals. If you are writing an application that will run on different operating systems or different hardware platforms, should your application or program be POSIX conformant? Is your application going to be used in several countries and, if so, do you need to follow internationalization guidelines? Are there security issues that you need to be concerned about? All these questions and more need to be answered during the requirements and specifications phase.

During this phase, you also need to consider how you plan to do your development work. For example, what major tools will you use for linking, debugging, implementing, and testing? Do you plan to call routines and use common files? Do you need to use a source control utility? Which installation utilities will people use to install your application?

The second phase involves design. During this phase, you design the flow of the program, sketching out the various functions and how they will fit together. You can also determine whether you can use existing routines, system calls, or common files to perform various functions in the application.

During the implementation phase you set up your programming environment and choose the tools you will use to create and modify the source files. Other tasks you will do include analyzing source code and building the application.

The testing phase involves testing the application, debugging the code, and analyzing performance.

The final phase includes making the program available for installation.

The ULTRIX operating system contains a number of tools and system features to help you with each phase. Table 1-1 shows which ULTRIX tools and features address the programming needs in each phase.

Table 1-1: Programming Phases and ULTRIX

Phase	Tools/Features
Requirements and specifications	Standards Window Environments Internationalization Security
Design	Routines Libraries Common Files
Implementation	sccs vi, GNU emacs, ex, ed lint, grep, cxref trace, ctrace sed, time dbx, dxdb make, compilers threads
Testing	diff shell scripts pixie, prof, gprof
Maintaining	setld tar, pxtar

As you can see from the table, in many instances an ULTRIX system offers more than one tool to do a job. Deciding which tool to use, as well as which programming language, is your choice.

1.2 Specification and Design Considerations

When designing an application, you need to make certain decisions that depend on the nature of the application. ULTRIX provides a number of features and tools to help you create applications that are, for example, portable, internationalized, or window-oriented.

One of the primary design considerations concerns adhering to UNIX-environment standards and portability. If you plan to write an application to run on an ULTRIX system well as on other UNIX-based operating systems, you probably want to consider following X/Open Portability guidelines and POSIX standards. You might also want to avoid using extensions to the ANSI standards that apply to the programming language you are using.

Another consideration is the terminal environment in which your application will be used. If end users have workstations or window terminals, you might want to design your application to use window displays and menus for the interface rather than command lines. The ULTRIX Worksystem Software contains a toolkit, a User Interface Language, and a window manager to help you create window interfaces for your application.

You might also need to design your application so that it can be used in a variety of countries. The ULTRIX operating system contains an internationalization package that provides tools and functions to help you write software to be used by people working in different natural languages.

1.2.1 Standards

Use of programming standards enhances the portability of programs and applications. Standard-compliant code is independent of the hardware or even the operating system on which the program runs. Writing programs according to portability standards makes it easy for users to move between systems without major retraining. Some standards include internationalization concepts as part of program portability.

The following are the primary standards in the UNIX programming environment:

- ANSI
- ISO
- POSIX
- X/Open

The various ANSI standards apply to specific programming tools such as programming languages, networks and communications protocols, character coding, and database systems. Information on conformance and extensions to a particular ANSI standard appears in the documentation set for the particular language, network system, or database system.

The ULTRIX operating system is conformant with the ISO, POSIX, and X/Open standards. For the most part, these standards apply to programs coded in C.

The ULTRIX system provides tools that allow you to write programs that conform to the POSIX and X/Open standards. Writing standard conformant programs involves the following:

- Working in the System V shell
- Using standard-conformant header files and the standard-conformant function library
- Compiling your program in the standard-conformant environment

The System V shell (sh5) contains features that are implemented to follow the POSIX standard. You change your login shell to the System V shell by using the `chsh` command. This command modifies your entry in the system password file. *The Big Gray Book: The Next Step with ULTRIX* has an example of a shell script that invokes the System V shell. (In a distributed environment, you might need to have your system administrator change your entry in the distributed password database.)

The ULTRIX header files contain POSIX- and X/Open-conformant information. These definitions are conditional and depend on the definition of two preprocessor symbols. When the symbols are defined correctly, POSIX- or X/Open-conformant header information is included in your program. (Otherwise, the default ULTRIX header information is included.)

ULTRIX provides the `libcP` function library, which conforms to the POSIX and X/Open standards. To use the standard conformant library, you must link with it as well as to the Berkeley Software Distribution (BSD) library, `libc`. Some functions differ between the two libraries. (For information on the differences, see Table 5-8.)

For your program to include standard conformant header information and functions from the standard conformant function library, you must compile your program in the standard environment. For information on compiling in the standard environment, see Section 2.1.4

If your standard-conformant program fails to compile, you need to check your programming environment to make sure that the POSIX function library is installed on your system. Lack of the POSIX function library will cause your program not to compile.

Later chapters contain more information on creating POSIX- and X/Open-conformant programs. For details about POSIX on the ULTRIX operating system, see the *POSIX Conformance Document.* Information on the POSIX standard is contained in the *IEEE Standard Portable Operating System Interface for Computer Environments*, published by the Institute of Electrical and Electronics Engineers, Inc. For detailed guidelines that meet the X/Open portability and connectivity objectives, see the *X/Open Portability Guide* documentation set.

1.2.2 Window Environments

The ULTRIX Worksystem Software (UWS) environment includes DECwindows applications, the X User Interface (XUI), X programming libraries, and guidelines for creating applications with XUI-compliant interfaces. The XUI provides a rich development environment for creating window-oriented applications.

The XUI allows you to develop applications that have simple, consistent, graphics-oriented interfaces. The consistency feature is important because it enables users to transfer knowledge gained from using one application to another, thus reducing the amount of learning time required and increasing their productivity.

The two major components of the graphics environment are the Xlib and XUI Toolkit programming libraries. Xlib supplies low-level routines for performing basic graphic and window functions. The XUI Toolkit library contains high-level routines for creating and managing user interface objects such as menus, scroll bars, and buttons. Typically, applications call routines from both libraries.

Using XUI Toolkit routines simplifies the task of creating a window interface. For example, creating a menu with XUI requires one call to a single XUI Toolkit routine. Creating the same menu using Xlib would require many more calls and lines of code. Using XUI Toolkit routines helps ensure that your application interface conforms to the XUI style, which is designed to make applications easy to learn and use.

The XUI Toolkit also includes the XUI User Interface Language (UIL) compiler and XUI Resource Manager (DRM) routines, which enable you to create an entire interface with one library call. The UIL and DRM let you separate form from function so that you can specify and modify the interface without having to recompile the entire application.

Using XUI programming library routines, you can write applications that create and manage windows to display output and accept input. Generally, the user interface for such an application consists of a series of windows. XUI programming library routines enable you to organize and manage a hierarchy of windows.

For an overview of UWS, see the *Introduction to the ULTRIX Worksystem Software Environment.* The *Guide to Writing Applications Using XUI Toolkit Widgets* and the *Guide to the XUI Toolkit: C Language Binding* focus on creating window applications.

1.2.3 Internationalization

An internationalized application allows users to interact with that application in their own language. Such applications are also designed to reflect the culture of the users' region.

Conventions for representing data can vary from one country to another and from region to region within a single country. Data such as number representation, currency symbols, and date representation are different, depending upon the local culture. For example, if your application displays or accepts monetary data, you want your users to be able to read or enter that data according to their local customs. The sum of five thousand monetary units would appear differently in different countries:

- $5,000 (United States dollars)
- L. 5.000 (Italian lire)
- 5,000 Dr (Greek drachmae)

To meet these internationalization requirements, you need to create applications that make no assumptions about language, locals customs, or coded character sets. Data specific to the local culture is held separate from the program logic. You use run-time facilities to bind your application to the appropriate language message text.

The ULTRIX internationalization package consists of the following tools and files:

- Message catalogs and associated tools
- A special set of library routines
- Internationalized interface definitions of standard C library routines
- An announcement mechanism
- Language support databases
- An international compiler for each database

Another consideration for multicultural applications is international keyboard support. In the international environment, you often use characters that your local keyboard might not support. You can create characters that do not exist as standard keys on your keyboard by using compose sequences. (A compose sequence is a series of keystrokes that maps to a single character.) Using these sequences, you can create any character from the character set that your terminal or DECterm session (assuming you are using ULTRIX Worksystem Software) currently has available.

For details about the ULTRIX internationalization package, see the *Guide to Developing International Software*.

1.3 Major Software Development Tools

ULTRIX is compatible with a number of higher-level languages and includes tools for linking and debugging programs.

1.3.1 Languages That Run in the ULTRIX Environment

The chief language that the ULTRIX operating system supports is C. In fact, a C language compiler is bundled with the ULTRIX operating system. Languages that the ULTRIX operating system supports include the following:

- ULTRIX C (bundled with ULTRIX)
- DEC C
- VAX C for ULTRIX
- DEC Fortran
- VAX FORTRAN for ULTRIX
- Pascal for RISC

Generally, programs written in languages that run on VAX hardware are compatible between the ULTRIX and VMS systems, provided they contain no system-specific dependencies.

Other languages available through Digital include Ada, COBOL, and Lisp.

Table 2-1 lists the compilers, their associated commands, and their respective platforms.

1.3.2 Linkers

In most instances, you can use the compiler command to link separate program object files into a single executable program.

As part of the compilation process, most compilers call the linker, `ld`, to combine one or more object files into a single program object file. In addition, the linker resolves external references, searches libraries, and performs all other processing required to create object files that are ready for execution. The resulting object module can either be executed or can serve as input for a separate `ld` run. (You can invoke the linker separately from the compiler by issuing the `ld` command.)

ULTRIX allows you to create applications composed of source program modules written in different languages. In these instances, you compile each program module separately and then link the compiled modules together in a separate step.

See Chapter 2 as well as the documentation sets for the individual languages for detailed information on compiling and linking programs. For information on the `ld`(1) command, see the *ULTRIX Reference Pages.*

1.3.3 Debuggers

The primary debugging tool on the ULTRIX operating system is `dbx`. In the window environment, you use the `dxdb` debugger, which is part of the ULTRIX Worksystem Software product. In addition, ULTRIX provides `ctrace` and `lint`. The `ctrace` utility is a C program debugger; `lint` is a tool for checking syntax in C programs.

Other debugging tools include `error`, which inserts error messages from a compiler into the source files at each point where an error occurs; `gcore`, which creates a core image file of a running process; and `trace`, which traces the system calls made by a command.

The `dbx` debugger, the most comprehensive debugging tool in a nonwindow environment, is discussed in detail in this manual. For information on `dxdb`, see the *Guide to the dxdb Debugger* in the ULTRIX Worksystem Software documentation set. The other tools are discussed in this manual and in the *ULTRIX Reference Pages.*

1.4 Source File Control

An integral part of creating a software application is managing the development and maintenance processes. The ULTRIX operating system has the SCCS (Source Code Control System) utility to help you store application modules in a directory, track changes made to those module files, and monitor user access to the files.

SCCS on the ULTRIX operating system provides support similar to SCCS utilities on other UNIX systems. In addition, ULTRIX has an `sccs` preprocessor, which provides an interface to the more traditional SCCS commands.

SCCS maintains a record of changes made to files stored using the utility. The record can include information on why the changes were made, who made them, and when they were made. You can use SCCS to recover previous versions of files as well as maintain different versions simultaneously. SCCS is useful for application project management because it does not allow two people to modify the same file simultaneously.

The `sccs` preprocessor provides a user-friendly interface for the SCCS user. Some of the commands are intuitive; others allow you to combine two SCCS functions in a single `sccs` command.

To use SCCS, you first need to create SCCS directories and files. If you are designing a large application with several developers, it might be advisable to assign a project librarian to set up the directory and files, and then be responsible for maintaining them. The project librarian "owns" all the files in the directory, regardless of who created them, and therefore can manipulate the files at any time without needing superuser privileges.

Once you have set up the SCCS directory and have created SCCS files, you can use SCCS commands to manipulate those files in the following ways:

- Retrieve files for compilation

  Using the `sccs get` command, you can retrieve files from the SCCS directory to compile your application. Although you can specify an option to get a writable copy of the specified file, it is best to use the `sccs edit` command for making modifications.

- Retrieve files for editing

  You use the `sccs edit` command to retrieve files so you can modify them. This action reserves the files so that no one else can modify them while you are working on them. Once the files are in your own directory, you can use any available editor to make your changes.

- Merge changes into the stored SCCS file

  After you have made all the changes you want to the file, you use the `sccs delta` command to merge those changes into the SCCS-stored file. SCCS prompts you for a comment to store with the changes. You use this comment to describe the changes you made to the file.

- Get information about your SCCS files

  Several `sccs` commands give you information about SCCS files. The `sccs info` command tells you which files are currently being edited and the names of the users who have retrieved the files for editing. The `sccs check` command is almost identical to `sccs info`, however, it does not print a message if nothing is being edited. It returns a nonzero exit status if anything is being edited. You can use this status in an `install` entry in a makefile to abort the operation if any file is reserved for editing.

The `sccs delta 'sccs tell'` command is also similar to the `sccs info` command, except that it displays only the names of the files being edited. The `sccs what` command tells you which version of a program is being run on your system.

SCCS is helpful in creating new releases, restoring old versions, reverting to older versions, and selectively deleting older versions. You can also use the utility to audit changes, recover a corrupted edit file, and maintain different versions of the same application.

For more information on using SCCS and the `sccs` command, see the *Guide to the Source Code Control System* and the `sccs(1)` reference page.

1.5 Program Installation Tools

Once you have created your program or application, you might want to kit it so that it can be distributed easily to other users. The ULTRIX operating system has several utilities that you can use to install, remove, combine, validate, and configure programs and applications.

Software for ULTRIX systems consists of a hierarchical group of files and directories. If your application or program consists of more than one file, or even of more than one directory, you need to determine how the files and directories are grouped within the hierarchy. The `setld` installation process preserves the integrity of each product's hierarchy when it is transferred from the development system to a production system (that is, when the product is installed). The kitting process includes grouping the component files for the product into subsets, some of which can be installed at the option of the system administrator.

Using the `setld` utility and its related tools to install and manage software products on ULTRIX systems provides the following benefits:

- Installation security

  The `setld` utility verifies each subset immediately after it is transferred from one system to another to make sure that the transfer was successful. Each subset is recoverable in case you need to reinstall one that has been damaged or deleted.

- Flexibility

  With the `setld` utility, you can let users choose which optional subsets to install. Also, users have the option of deleting subsets and then reinstalling them later, as needed. You might use this feature to provide multiple language support for your application or to allow users to select among optional features of your application.

- Uniformity

  The `setld` utility is an integral part of the ULTRIX installation implementations. Using this utility to prepare and install software kits for your application enhances compatibility with future ULTRIX installation architecture. In addition, kits produced with `setld` can be loaded on a server machine for installation over the network using the Remote Installation Services (RIS) utility or the Diskless Management Services (DMS) utility.

Using `setld`, you can load your application on any of the following distribution media for installation on other systems:

- Data disks, such as RA60 disk packs or CDROM optical discs
- TK50 tapes
- MT9 tapes

1.5.1 Utilities for Creating setld-Compatible Kits

In order for people to use the `setld` utility to install your application, you must create your kit so that it is compatible with `setld`. There are two ULTRIX utilities that you use to create such kits:

- `newinv`

  The `newinv` utility processes a master inventory input file. The output of the `newinv` utility is a file that has the current master inventory of the software product. This file contains a list of all the files that make up your application and tells which subset each file belongs to.

- `kits`

  The `kits` utility produces subset images, inventories, and control files from the input files that have been transferred from your source directory. The utility also generates data files that make up the media master in the output directory.

Information about using the `newinv` and the `kits` utilities is located in the *Guide to Preparing Software for Distribution on ULTRIX Systems.*

1.5.2 Additional Installation Options

If you want to have your program or application installed remotely or into a diskless environment, you need to plan the file configuration up front. The ULTRIX operating system contains the Remote Installation Services (RIS) utility to enable users to remotely install software and the Diskless Management Services (DMS) utility to enable users on a client machine to use software installed in a special DMS area on a server machine.

The RIS utility performs remote installation services. These services allow users to install software on a client machine through the TCP/IP local network. Both the server and the client can be either a RISC or a VAX machine running the ULTRIX operating system.

The DMS utility allows users to install products into a diskless management services area on a server machine and register diskless clients. Once a client machine is registered, users on that client can access the software on the server machine without having it installed on the client machine. Both the server and client can be either a RISC or a VAX machine running the ULTRIX operating system.

If you want to prepare your program or application for use with either RIS or DMS, refer to the documentation on each utility. For information on RIS, see the *Guide to Remote Installation Services*; for information on DMS, see the *Guide to Diskless Management Services.*

Compiling, Linking, and Building Programs 2

This chapter describes the components of the compiler system, how to use them, and how to build programs using an automated method. This chapter discusses the following topics:

- Compiler commands, sometimes called driver programs, examine the command line for options and files, and pass the appropriate options and files to the various components (such as preprocessors and compilers). Thus, a driver program controls which of the other components are run.
- Preprocessors may be run before the appropriate compiler. For example, the C language preprocessor is the `cpp` preprocessor.
- Compilers (or the assembler) read one or more source files and create an object module (usually a temporary file) for the linker.
- The Archiver stores either object (coff) files in an archive object library or ucode files in an archive ucode library.
- The Linker (link editor) reads one or more object files and creates the executable program.
- You can build an application in an automated way by creating makefiles that you process using the `make` command.

You create and modify source programs using the text editor of your choice, such as `vi`.

2.1 Compiling Using Driver Programs

Each language compiler has its associated compiler command (such as `cc`, `c89`, or `f77`) , which in turn invokes the appropriate driver program. When you type the appropriate command followed by the appropriate options and at least one file name, the driver program examines the specified options and the suffix of each file name to determine which preprocessor or compiler (or the assembler or linker) is to process each file.

For instance, a file named `main.c` is assumed to be a C language program to be processed by the C language preprocessor (`cpp`) and C compiler; a file named `test.p` is similarly assumed to be a Pascal program.

In the ULTRIX programming environment, a single compiler command can compile and link the source file to create an executable program. In addition, if multiple source files have been specified, files may be passed to other compilers before linking. When you type a compiler command, the driver program can perform multiple actions:

- Based on the file name suffix of each file, the driver program decides whether to call the appropriate preprocessor, compiler (or assembler), or linker. You can specify command options to prevent linking or to prevent or request preprocessing.

- The default behavior is that source files are automatically linked together if compilation (or assembly) is successful. You can specify the `-c` option to prevent linking (and thus prevent creation of the executable program) and retain the `.o` object file for a subsequent link operation.
- The linker creates an executable program file with a default name of `a.out`. You can use the `-o` *name* option to specify a name other than `a.out`.
- In most cases, to run an executable program in your current working directory, you only need to type its file name. To run the program `a.out` located in your current directory (if your current directory is in your path), type:

  ```
  % a.out
  ```

  If the executable program is not in a directory in your path, type the directory path before the file name or type:

  ```
  % ./a.out
  ```

The compiler commands invoke the driver programs that compile, optimize, generate object code, and link your programs. Each driver program knows the appropriate libraries associated with the main program (most include `libc.a`) and passes those libraries to the linker.

The linker is usually accessed using one of the compiler commands instead of the `ld` command, even if you need to link only object files.

Unlike the compiler commands, the assembler (`as`) can only assemble a single file, which is assumed to contain assembler code (the suffix is ignored). The `as` command does not automatically link the assembled object file. Thus, if you use the assembler, you need to use a separate `ld` command.

Table 2-1 shows the compilers available for use on ULTRIX systems, whether they are part of the ULTRIX operating system, and on which platforms they apply. (For a more recent list of available products, contact a Digital sales representative.)

Table 2-1: Compilers Available for RISC and VAX Processors

Compiler Command	Language	Included with ULTRIX Kit for:	Layered Product for:
`as`	Assembly	RISC, VAX	
`cob`	COBOL		RISC
`cc`	C (pcc)	RISC, VAX	
`c89`	C (DEC C)	RISC[a]	
`f77`	FORTRAN		RISC
`fort`	FORTRAN		VAX
`lisp`[b]	Lisp		RISC
`pc`	Pascal	VAX	RISC

Table note:

a. DEC C (`c89` command) currently (Version 4.2) must be separately ordered (for media cost), but is included in the ULTRIX license. It provides ANSI C compatibility.

b. Lisp uses the `lisp` command to start up the Lisp environment, allowing you to use other Lisp commands to compile files and so forth. Lisp does not pass information to `cc` and does not invoke the assembler.

2.1.1 Compiler Command Input and Output Files

Most driver programs recognize an input file by its filename suffix. Table 2-2 lists the valid suffixes for languages available for the RISC and VAX platforms.

Table 2-2: Compiler Command Input and Output File Suffixes

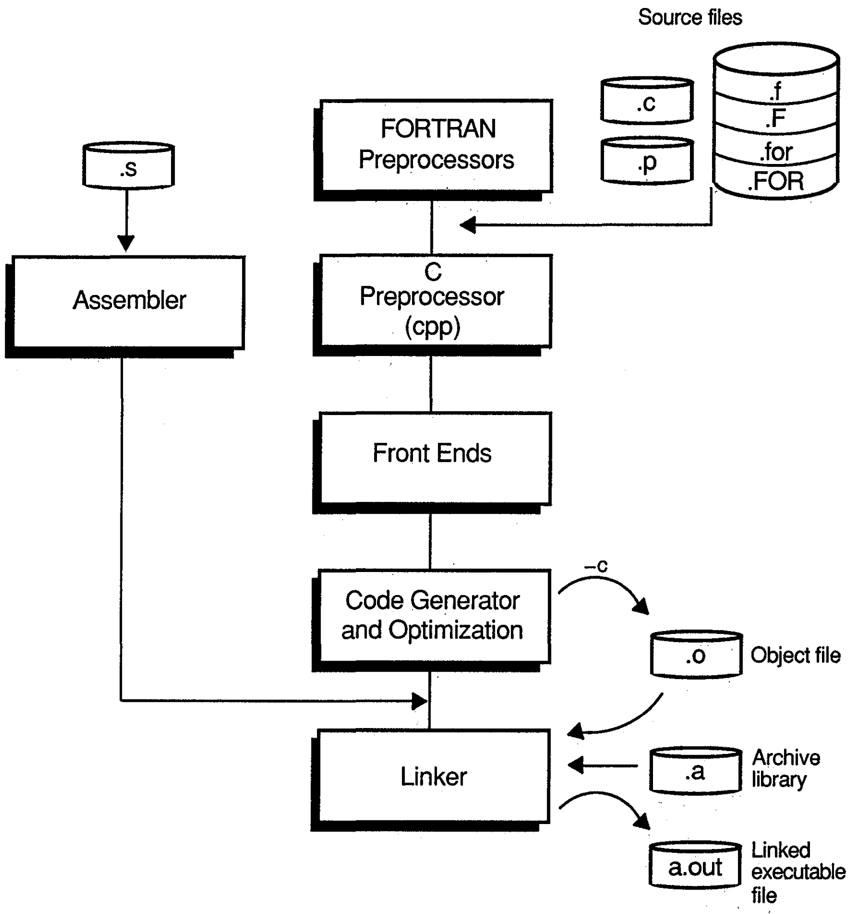
Suffix	Description
.a	Object archive library.
.c	C source file.
.e	`efl` source file.
.f, .for, .F, .FOR	FORTRAN source file.
.o	Object file.
.p	Pascal source file.
.r	`ratfor` source file.
.s	Assembly source file.
	RISC Specific
.b	ucode object library. Not all RISC compilers can generate ucode files.
.cob, .cbl, .CBL	COBOL source file.
.i	An intermediate file created by preprocessor execution (before compilation) in the source language of the processing driver. For example: `pc -c` *`source`*`.i` In this case, the `pc` command assumes that *`source`*`.i` contains Pascal source code.
.lsp, .lisp	Lisp source file.
.u	ucode object file. Not all RISC compilers can generate ucode files.

2.1.2 Components of the Compiler System

When you compile a program, you usually select one or more options that affect debugging, optimization, and profiling facilities, as well as the names assigned to output files.

Figure 2-1 illustrates the relationship between the major components of the compiler system and their primary input and output files for RISC driver programs and the `fort` command driver on the VAX platform. (Figure 4-1 provides additional detail on the use of RISC platform ucode files and optimization.)

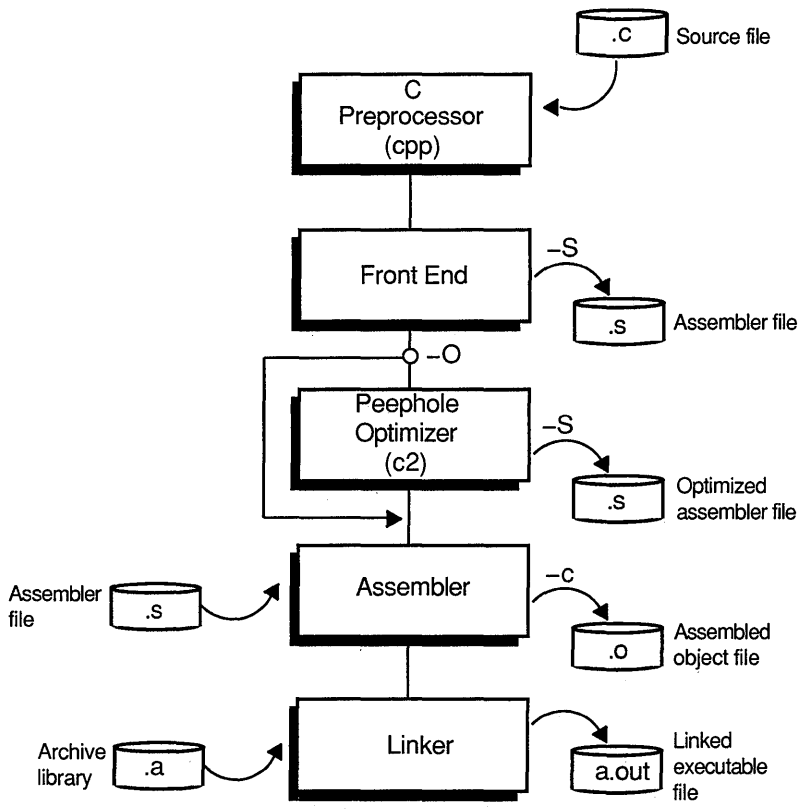
Figure 2-1: Major Compiler Phases



ZK–0277U–R

Figure 2-2 illustrates the relationship between the major components of the compiler system and their primary inputs and outputs for all VAX driver programs except the `fort` command driver.

Figure 2-2: Compiler Phases Used by Most VAX Driver Programs

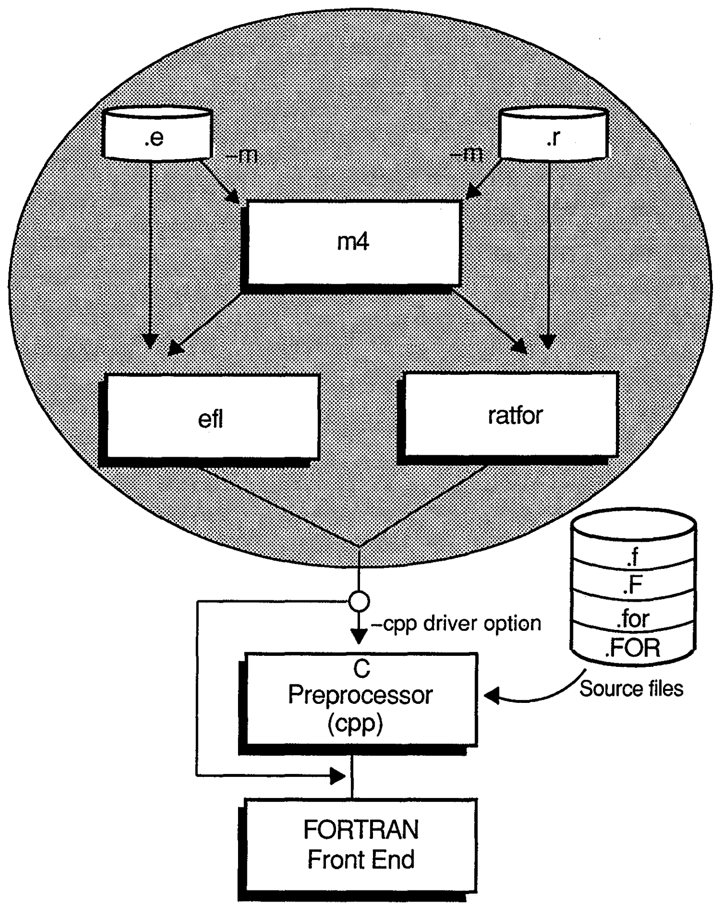


ZK-0177U-R

Note that FORTRAN on both the RISC and VAX platforms use preprocessors that the other languages do not use.

Figure 2-3 illustrates the relationship of the FORTRAN preprocessors.

Figure 2-3: The FORTRAN Preprocessors



ZK–0062U–R

Some options have defaults. For example, the default name for object files is `*filename*.o`

The specified *filename* is the name of the source file without its filename suffix. The default name for an executable program file is `a.out`.

The following example shows compilation of two C source files, `main.c` and `sub.c`, that generates an executable program file. The following command invokes the compiler:

```
% cc main.c sub.c
```

The C compiler compiles the source files (`main.c` and `sub.c`), creates one or multiple object modules (depending on the compiler), which are deleted after linking, and a single executable program, `a.out`.

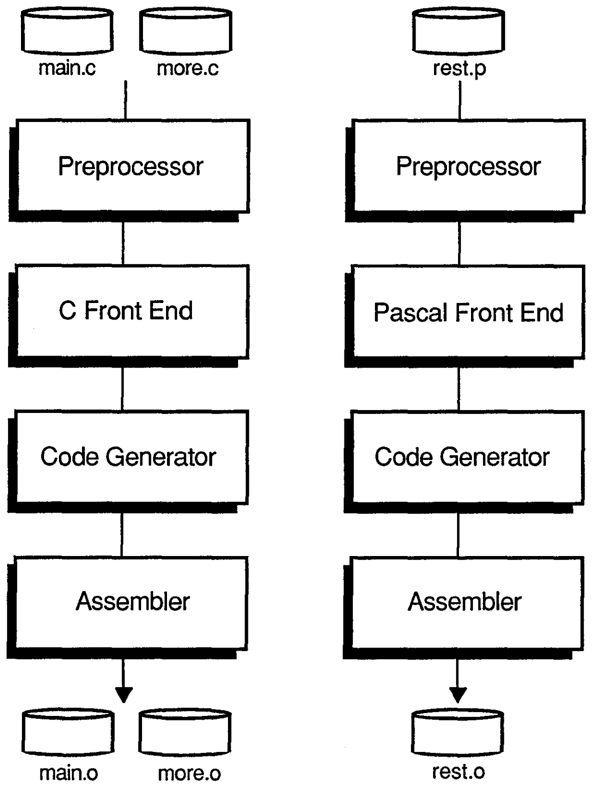
2.1.3 Compiling Multilanguage Programs

For a very large application, it may be easiest to perform incremental compilation and subsequent linking of the application. When the source language of the main program differs from that of a subprogram and neither language is C, you may need to compile each program module separately with its respective driver and then link them in a separate step. In either case, you can create objects suitable for linking by specifying the `-c` option, which stops the driver after it creates the object file. For example:

```
% cc -c main.c more.c
% pc -c rest.p
% pc main.o more.o rest.o
```

Figure 2-4 illustrates the compilation control flow for these commands.

Figure 2-4: Compiling Multilanguage Programs



ZK-0063U-R

Most language driver programs pass information to `cc`, which, after processing, passes information to `ld`. When one of the modules to be compiled is a C program, you can usually use the driver command of the other language to compile and link both modules. In most cases, if the driver command invokes `cc`, such as the FORTRAN and Pascal compilers, use the command driver associated with that language to make sure the correct compiler is invoked and that the correct libraries are passed to the linker.

For instance, if you have a FORTRAN main program `main.f` that calls a C function contained in `syscall.c`, you could use the `f77` (RISC) command or the `fort` (VAX) command to compile and link both modules. For example:

```
% f77 main.f syscall.c
```

2.1.4 Compiling in the POSIX or X/Open Environment

As mentioned in Chapter 1, the ULTRIX system allows you to write programs that conform to the POSIX or X/Open standards. When you write standards-conformant programs, you must compile your program in the POSIX or X/Open programming environment. You can compile your program in one of these environments using one of two methods:

- Set an environment variable and preprocessor symbols before you issue the `cc` or `c89` command
- Set the environment variable and preprocessor symbols on the `cc` or `c89` command line.

Follow these steps to set the environment variable and preprocessor symbols before you issue the `cc` or `c89` command:

1. Define the PROG_ENV variable. For example:

   ```
   % PROG_ENV=POSIX; export PROG_ENV
   ```

   You must set the PROG_ENV variable to POSIX when you write POSIX- or X/Open-conformant programs.

2. Create a local header file that defines the the _POSIX_SOURCE or _XOPEN_SOURCE preprocessor symbol. (You can also define the preprocessor symbols directly in your source file.)

   Define only the _POSIX_SOURCE symbol if you are writing POSIX-conformant programs. Define both _POSIX_SOURCE and _XOPEN_SOURCE if you are writing X/Open-conformant programs.

   The following example shows a local header file that defines the _POSIX_SOURCE and _XOPEN_SOURCE preprocessor symbols:

   ```
   #define _POSIX_SOURCE
   #define _XOPEN_SOURCE
   ```

3. Include the local header file in your source program. (If you define the preprocessor symbols directly in your source file, skip this step.) Place the `include` directive for the local header file before any `include` directive for an ULTRIX header file.

   For example, if you name the local header file `standard_head.h`, use the following directive in your source program:

   ```
   # include "standard_head.h"
   # include "stdio.h"
   ```

   Be sure to include the local header file in each source file for your program.

4. Compile your program using the `cc` or `c89` command.

   For example, suppose you are writing an X/Open conformant program and your program consists of three modules named `main.c`, `more.c`, and `rest.c`. To compile your program, issue the following command:

```
% cc main.c more.c rest.c
```

When you compile a program in the POSIX environment and the _POSIX_SOURCE and _XOPEN_SOURCE symbols are defined, the `cpp` or `cpp89` preprocessor includes X/Open conformant header information in your program. The `ld` linker includes standard conformant functions in your program image.

You can define the _POSIX_SOURCE or _XOPEN_SOURCE symbol and the PROG_ENV variable on the `cc` or `c89` command line, using the `-D` and `-Y` command options. These options allow you to avoid modifying source code to define a preprocessor symbol and issuing commands to define the PROG_ENV variable. The following example uses `cc` command options to define _POSIX_SOURCE and PROG_ENV:

```
% cc -D_POSIX_SOURCE -YPOSIX main.c more.c rest.c
```

In this example, the `-D` option defines the _POSIX_SOURCE symbol to the value 1. The `-Y` option sets the programming environment to POSIX. These definitions are in effect only during the execution of the `cc` command. For more information on the `-D` and `-Y` options, see the `cc`(1) and `c89`(1) reference pages.

2.1.5 Using error with Compiler Driver Programs

When the compiler issues a diagnostic message indicating that your source code contains an error, you might want to see the error displayed beside the source line that caused the error. You can use the `error` command to take errors from the `cc` command and insert those messages into your source file at the point the error occurred. In addition to the `cc` command, `error` supports `as`, `ccom`, `cpp`, `f77`, `ld`, `lint`, `make`, `pc`, and `pi`.

You normally run the `error` command with the language processor connected through a pipe to its standard input. Some language processors write error messages to standard output; others write messages to standard error. To be sure error message are passed to `error`, pipe both standard output and standard error into the `error` command. If an error message refers to more than one line in a source file, `error` duplicates the messages and inserts it before each appropriate line.

The `error` command has the following syntax:

[*language_processor* **|&**] **error** [*options*]

For complete information on the options to the `error` command, see `error`(1) in the *ULTRIX Reference Pages.*

The following example attempts to use the `cc` command to compile a program named `sample.c`. The output from `cc` is sent to the `error` command.

```
% cc sample.c |& error

        2 non specific errors follow
[unknown] ese if ((tmp = getenv("TEXT")) != 0)
[unknown] ese if ((tmp = getenv("TEXT")) != 0)
1 file contains errors "sample.c" (1)

File "sample.c" has 2 error.
        2 of these errors can be inserted into the file.
You touched file(s): "sample.c"
```

The `error` command inserts the error message as a comment in the source file. The comment has the following format:

/*### [*error message text*] %%%*/

Editing `sample.c` and searching for "/*###" reveals the following error:

```
    str[0]=' '
    if(argc > 1)
            strncpy(str,*++argv,MAX);
/*###43 [cc] Error: syntax error%%%*/
/*###43 [cc] Error: ese undefined%%%*/
        ese if ((tmp = getenv("TEXT")) != 0)
               strncpy(str,tmp,MAX);
```

In this case, the source code line flagged by the error messages contains an "else" clause that is missing a letter.

2.2 Using the C Preprocessor

The C preprocessor (`cpp`) is invoked by default for `.c` and `.p` files by most driver programs. With FORTRAN, each file with a `.F` suffix causes `cpp` to be automatically invoked for that file. For `.f`, `.for`, and `.FOR` files, `cpp` is not automatically invoked when used with the `fort` (VAX) or `f77` (RISC) command. Except for the assembler, most driver commands allow you to invoke `cpp` by using the `-cpp` option on the driver command line.

If you use the `c89` command, the C preprocessor is `cpp89`, which can be separately invoked.

2.2.1 Including Common Definition Files

When you write programs, you often have common definition files that you share among a program's modules. These files usually define known constants, declare types (routines types or data types, including data structures), and declare function prototypes (such as library functions or system services). Definition files, called `#include` or header files in the C programming language, let you share common information between files in a program. These header files typically have a `.h` suffix.

Most supported languages allow you to include these files in your program's source code using the C preprocessor, but if a header file contains C code, include it only from a C language program.

If you intend to debug your program using the `dbx` debugger, do not place executable code in an include file. The debugger recognizes an include file as one line of source code; none of the source lines in the file appears during the debugging session.

To specify an include file in your program, begin the `#include` directive in column 1 of your source file. There are two forms of the `#include` directive, where the file to be included is specified using double quotation marks or angle brackets, as follows:

```
#include "file1"
#include <file2>
```

Each file name listed in this manner indicates the name of the include file. Because the name of the first include file is in double quotation marks, the C preprocessor searches for them first in the directory where the source file is located and then searches the default directory, `/usr/include`. Because the names of the next include file is enclosed in angle brackets, the C preprocessor searches for them only in the default directory, `/usr/include`. You can specify the pathname before the filename in the `#include` directive.

You can also use the `-I`*`dir`* option to specify additional pathnames (directories) to be searched by the C preprocessor for `#include` files. The C preprocessor searches first in the directory where the source file resides, followed by the specified pathname *`dir`*, and then the standard directory `/usr/include` . For most compiler commands, if *`dir`* is omitted, the standard directory `/usr/include` is not searched. The `c89` driver allows the `-I`*`dir`* form as well as the form `-I` *`dir`* (a space between the `-I` and *`dir`*) for POSIX compatibility.

If you want to prevent the `c89` driver from searching the `/usr/include` directory, you must specify the `-I` option either directly before another option or at the end of the command line. When the `c89` driver sees `-I` on the command line followed by a space and then a hyphen (-) or the end of the line, the driver interprets the option as `-I` (directory `/usr/include` is not searched). If the characters following are neither a hyphen nor the end of the line, `c89` interprets those characters as the additional pathname to search, *`dir`*.

2.2.2 Setting Up Shareable Include Files in RISC Programs

For the RISC architecture, C, Pascal, FORTRAN, and assembly source code can reside in the same `#include` files and then can be conditionally included in programs as required. To set up a shareable include file, create a `.h` file and conditionalize the respective code as follows:

```
#ifdef LANGUAGE_C
       .
       .
       .
#endif
#ifdef LANGUAGE_PASCAL
       .
       .
       .
#endif
#ifdef LANGUAGE_FORTRAN
       .
       .
       .
#endif
#ifdef LANGUAGE_ASSEMBLY
       .
       .
       .
#endif
```

2.3 Creating Archive Libraries

An archive library is a file that contains one or more routines in object (`.o`) or ucode (`.u`) file format. (Only RISC systems support the ucode file format and not all RISC compilers produce ucode files.) When a program calls an object or ucode file not explicitly included in the program, the linker looks for that object in an archive library. The linker then loads only that object (not the whole library), and links it with the calling program. For more information about linking with archive libraries, see Section 2.4.3.

To create an archive library, you must first create object files or ucode files. Use a compiler command to create object files from your source file. For example, the following `cc` command creates an object file from the source file `main.c`:

```
% cc -c -o main.o main.c
```

To create a ucode file, use a compiler command similar to the following:

```
% cc -j main.c
```

This command creates a file named `main.u`.

Use the `ar` archiver to create and maintain archive libraries that contain your object or ucode files. The `ar` archiver performs the following tasks:

- Copies new files into the archive library
- Replaces existing files in the library
- Moves files within the library
- Extracts individual files from the library into individual object or ucode files
- Displays information about files in the archive library

To execute the `ar` command, use the following syntax:

ar *options* [*posname*] *archive file...*

You name the archive you want created or modified in the *archive* argument. The *file* argument names the object or ucode files you want the archiver to use. You can name a number of object or ucode files.

When you are inserting or moving files in the archive, you can determine their position using the *posname* argument. In the *posname* argument, you name an existing file in the archive. You then use command options to specify adding a file before or after the file you specify in *posname*.

The following shows an example of using the `ar` command:

```
% ar -r libfft.a main.o
```

This command specifies adding the `main.o` file to the end of the `libfft.a` archive.

You should also run `ranlib` on a archive library to add a table of contents for linking purposes, such as:

```
% ranlib libfft.a
```

For more information about the `ar` and `ranlib` commands, including descriptions of the options you can use, see `ar(1)` and `ranlib(1)` in the *ULTRIX Reference Pages.*

2.4 Linking Files

The linker (`ld`) combines one or more object files (in the order specified) into one executable program file, performing relocation, external symbol resolutions, and all other processing required to make object files ready for execution. Unless you specify otherwise, the linker names the executable program file `a.out`. You can execute the program file or use it as input for another linker operation.

The linker supports all the standard command line features of other UNIX system linkers (except System V command language files, which contain a description of a load module).

For further information about the linker, see `ld(1)` in the *ULTRIX Reference Pages.*

2.4.1 Linking Using the Compiler Commands

You can use a compiler command instead of the `ld` command to link separate objects into one executable program. Because the compiler driver programs automatically pass the libraries associated with that language to the linker, using the compiler command is usually recommended. You can also specify additional libraries to be searched for unresolved references using the `-l` option.

Depending on the nature of the application, decide whether to compile and then separately link or to perform both compilation and linking using one compiler command. Factors to consider include whether all source files are in the same language, whether all source files are readily available, the number of object files, and so forth.

One reason to compile and link modules with a single command is when you want to optimize your program. Most compilers support increasing levels of optimization with the use of certain options. For example, the `-O0` option requests no optimization (usually for debugging purposes), while the `-O1` option requests certain local (module specific) optimizations. On RISC systems, you can request cross-module optimizations by using the `-O3` or `-O4` option. (These options are valid only when the multiple modules you are compiling are written in the same language.) In this case, compiling and linking in one operation allows the compiler to perform the maximum possible optimizations.

Certain compilers may provide a combination of options (such as `-c` and `-o` *`name`*) that allow multiple source files to be compiled into a single object module, which allows the interprocedural optimizations to occur, yet retains the object file (see your language documentation).

Each compiler command (except the assembler) recognizes the `.o` suffix as the name of a file that contains object code suitable for linking and immediately invokes the linker. You could link object modules using the `pc` Pascal driver, as follows:

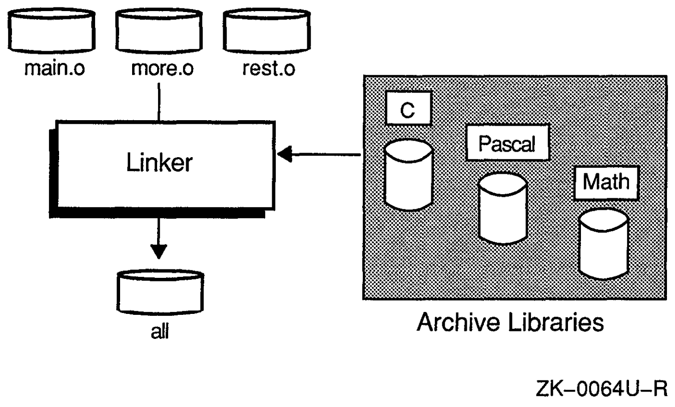
```
% pc -o all main.o more.o rest.o
```

This command produces the executable program object of the specified name, `all`. You could achieve the same results using the `cc` compiler command, as follows:

```
% cc -o all main.o more.o rest.o -lp -lm
```

Figure 2-5 illustrates the control flow for the `pc` and `cc` commands used in these examples.

Figure 2-5: Example of pc and cc Driver Control Flow



Note that to link the appropriate libraries with the `cc` driver, you must specify two additional options that the `pc` driver uses by default: the `-lp` option, which specifies the Pascal link library, and the `-lm` option, which specifies the math link library. Both `pc` and `cc` use the C library `libc.a` by default.

For information about the default libraries used by each compiler command, see the appropriate command in the *ULTRIX Reference Pages*, such as `cc`(1), and the reference pages for layered products.

2.4.2 Linking Using the ld Command

The `ld` command is usually only used with object modules created by the assembler. The `ld` command has the following syntax:

ld *options object ...*

Because the `as` assembler does not automatically invoke the linker, to link a program written in assembly language, do either of the following:

- Assemble and link by using one of the other compiler commands (for example, `cc`). The `.s` suffix of the assembly language source file automatically causes the compiler command to invoke the assembler.
- Assemble by using `as`; then link the resulting object file by using `ld`.

For further information about the options and libraries that affect the linking process, see `ld`(1) in the *ULTRIX Reference Pages.*

2.4.3 Specifying Libraries

When you compile your program on the ULTRIX system, it is automatically linked with the C library, `libc.a`. If you use routines that are not in `libc.a` or one of the other archive libraries associated with your compiler command, you must explicitly link your program with the library. Otherwise, your program will not be linked correctly. This section explains three situations in which you need to explicitly specify libraries and the options you use to specify the libraries.

If you compile multilanguage programs, be sure to explicitly request any required run-time libraries to handle unresolved references. You load the libraries by specifying the `-lstring`, where *`string`* is an abbreviation of the library name.

For example, if you write your main program in C and some procedures in Pascal, you must explicitly specify the `libp.a` Pascal library and the `libm.a` math library by specifying the `-lp` and `-lm` options. When you use these options, the loader replaces the `-l` with `lib` and adds the specified character (p or m) and the `.a` suffix. Then, it searches the following directories for the resulting library name (in this case `libp.a` and `libm.a`):

- `/lib`
- `/usr/lib`
- `/usr/local/lib`

For a list of the libraries that each language uses, see the reference pages for the appropriate command.

You may need to specify libraries when you use UNIX system packages that are not part of a particular language. The reference pages for these packages list the required libraries. For example, the plotting subroutines require the libraries listed in the `plot`(1g) reference page.

If you store object or ucode files in an archive library, you must include the pathname of the library on the compiler or loader command line. For example, the following command specifies that the `libfft.a` archive library in the `/usr/jones` directory is to be loaded along with the Pascal library `libp.a`:

```
% cc main.o more.o rest.o /usr/jones/libfft.a -lp
```

The linker searches libraries in the order you specify. Therefore, if any file in your archive library uses data or procedures from the Pascal library, you must specify the archive library line before you specify the Pascal library.

Using ucode object libraries is similar to using other object libraries. To load from a ucode library, specify the `-kl` *x* compiler option or ucode loader option. The following example loads a file from a ucode library:

```
% cc -klucode_lib -o output main.u more.u rest.u
```

Because the libraries are searched as they are encountered on the command line, the order in which you specify them is important. If a library is made from both assembly and high-level language routines, the ucode object library contains code only for the high-level language routines. Unlike a coff object library, the ucode library does not contain code for the routines. In this case, you must specify to the ucode loader both the ucode object library and the coff object library, in that order, to ensure that all modules are loaded from the proper library.

If the compiler driver is to perform both a ucode load step and a final load step, the object file created after the ucode load step is placed in the position of the first ucode file specified or created on the command line in the final load step.

2.4.4 Linker Options

Table 2-3 describes some of the more frequently used linker options that are available with most driver programs.

Table 2-3: Linker Options

Linker Option	Description
`-L`*`dir`*	Specifies the pathname *`dir`* as an additional search directory for the linker, which is searched for unresolved global references before the directories associated with the driver command used. For those driver programs that use the `ld` linker, the directories searched are `/lib`, `/usr/lib`, and `/usr/local/lib`. The `c89` command allows a space between the `-L` and *`dir`* (in the form of `-L` *`dir`*) for POSIX compatibility. Other compiler commands allow the use of `-L` as an option to indicate not to search in the standard directories.
`-l`*`string`*	Specifies additional libraries to be searched, in addition to the libraries associated with the driver command used. The characters specified as *`x`* are added to `lib` and form a file name of a library. For example, if you specified `-lm`, the linker searches for `libm.a`. This option can be repeated in the desired search order to specify multiple additional libraries. The `c89` command allows a space between the `-l` and *`string`* (in the form `-l` *`string`*) for POSIX compatibility.
`-o` *`name`*	Specifies a file name *`name`* be given to the executable program rather than the default, `a.out`.
	RISC Specific
`-bestGnum`	Requests that `ld` calculate an efficient value for the data size limit of the global pointer area. The calculated value can be used as *`num`* for the `-G` *`num`* option in a subsequent link operation.
`-G` *`num`*	Specifies *`num`*, in bytes, as the limiting size of global data items to be included in the global pointer area for this link operation. The default value is 8 bytes. The more data items placed in the global pointer area, the faster the program executes. However, if too many data items are included because of a high *`num`* value, the total size of all data items below the *`num`* value could exceed the fixed size of the global pointer area (65,536 bytes), resulting in a link error. If this error occurs, reduce the specified *`num`* value.

2.5 Building Programs with the make Program

The `make` program keeps track of the many files that compose a large program. Information about file dependencies and how files are to be processed is stored in a makefile. Thereafter, the `make` command processes only those files that have been changed — or depend on files that have been changed — since the last `make`. Using `make` ensures that whenever a program is rebuilt, only the required processing is performed; files that do not need to be recompiled or relinked are not.

For more information about the `make` program, see `make(1)` in the *ULTRIX Reference Pages*, and the article *Make—A Program for Maintaining Computer Programs* in the *ULTRIX Supplementary Documents, Volume 2: Programmer.*

Makefiles are text files you create and edit. Makefiles contain two types of lines:

- Statements of dependencies, such as *file A depends on files B and C*
- Commands to be executed, such as `c89 -o A B.o C.o`

Consider a program called `big_program` that is made from three C modules: `x.c`, `y.c`, and `z.c`.

- Module `x.c` includes `project.h`
- Modules `x.c` and `y.c` include `stdio.h`
- Object file `y.o` uses the user-supplied library `projlib.a`
- Object files `y.o` and `z.o` use the object library `/lib/libcurses.a`.

The makefile for `big_program` is shown in Example 2-1.

Example 2-1: Makefile

```
# The executable file depends on object files
1 all : x.o y.o z.o  # Comments are ended by the end of the line
2       c89 -o big_program x.o y.o z.o -lcurses -lprojlib
                               # This command is executed if x.o, y.o, z.o,
                               # or any of the libraries change
# The object files depend on source files and headers
  x.o : x.c project.h  # x.o depends on x.c and project.h
3       c89 -c x.c     # This command is executed if x.c
                       # or project.h change

  y.o : y.c stdio.h  # y.o depends on y.c and stdio.h
        c89 -c y.c   # This command is executed if y.c or stdio.h change

  z.o : z.c          # z.o depends on z.c
        c89 -c z.c   # This command is executed if z.c changes
```

Each line that issues a command must begin with a tab.

The numbers in the following list correspond to the numbers in Example 2-1:

$\boxed{1}$ The name before the colon is called a target. It depends on everything after the colon; in this case, three object files. A target can be a name created for convenience (as `all` has been). Any target can be specified on the `make` command line.

$\boxed{2}$ This line links, but does not compile (because the files end with `.o`). This example command line includes both the `curses` and `c` libraries.

$\boxed{3}$ This line compiles, but does not link (because of the `-c` option) `x.c`, which includes `stdio.h` with an `#include` preprocessor directive.

The makefile in Example 2-1 can be written a shorter way, as shown in Example 2-2. The first character of every command line still must be a tab.

Example 2-2: Shorter Makefile

```
  all : x.o y.o z.o
1       c89 -o big_program x.o y.o z.o -lcurses -lc

2 x.o y.o : stdio.h
```

The numbers in the following list correspond to the numbers in Example 2-2:

1 This line is identical to the one in the previous example.

2 This is the only dependency line needed because make assumes x.o, y.o, and z.o depend on C source files of the same name but with a .c ending. The only other information needed is that x.o and y.o depend on stdio.h.

Each time a component of big_program changes, all that must be done to create a new executable file is issue the make command. The make program compares the times that dependent files were last changed, then compiles and links only the files that must be. The make command searches the current directory, first for a file named makefile, then for a file named Makefile. Makefiles with other names can be specified with the -f option; for example:

```
% make -f Big_prog_makefile
```

Any name to the left of a colon in a makefile is a target. If no target is specified on the make command line, the first target in the makefile is used. In Example 2-2, the default target is all. In the following command, the target is y.o:

```
% make y.o
```

2.5.1 make Macros

The make command has its own macros that can be defined within makefiles. These macros allow groups of objects to be handled by a single name, as shown in Example 2-3. A macro name is a string to the left of an equal sign; whatever is to the right of the equal sign is the macro's expansion. Macros are invoked by preceding the macro name with a dollar sign and left parentheses, and following the name with a right parentheses; for example: $(MACRO).

Example 2-3: Makefile with Macros

```
# makefile for big_program rewritten using macros

C89_OPTIONS = -g -std       # A macro for compiler options

OBJECTS = x.o y.o z.o       # A macro for object files

LIBS = -lcurses -lc         # A macro for libraries

big_program : $(OBJECTS)    # big_program depends on OBJECTS
        c89  $(C89_OPTIONS)  $(OBJECTS)  $(LIBS)

x.o y.o : stdio.h
```

Macros can be redefined on the make command line; for example, the following command changes the c89 options:

```
% make "C89_OPTIONS = -g -std -check"
```

2.5.2 Performing Other Tasks with make

Targets need not be file names; they can be any word, as shown in Example 2-4. Therefore, targets can lead to groups of commands that perform tasks other than compiling and linking.

Example 2-4: Makefile with Command Targets

```
# This makefile can run lint on source files, print source files
# that have changed since the last printing

SOURCE = x.c y.c z.c

OBJECTS = x.o y.o z.o

LIBS = -lcurses -lc

big_program : $(OBJECTS)
        c89  -o big_program  $(OBJECTS)  $(LIBS)

x.o y.o : stdio.h

clean : rm *.o              # The command "make clean" removes all
                            # object files from the working directory

lint : $(SOURCE)            # If any source file has changed,
       lint $(SOURCE)       # run lint on all the source files.
       touch lint           # Update the time of the most recent lint run

print : $(SOURCE) makefile  # If makefile or any source file has changed,
        lpr $?              # print files changed since the last printing.
        touch print         # Update the time of the most recent printing
```

In Example 2-4, `clean`, `lint`, and `print` are targets that cause `make` to execute shell commands. Any shell commands can be used. The `make`-defined macro `$?` stands for dependency names that are younger than the target. Consider the line below the `print` target: `lpr $?`; this line prints the makefile and every file in the SOURCE macro (`x.c y.c z.c`) if they are younger than the `print` target. In other words, a file is printed if it has been changed since `make` with the `print` target was last run.

The preceding makefile could be used with the following command to run `lint` on source files changed since the last running of `make` with the `lint` target:

```
% make lint
```

The following command specifies multiple targets:

```
% make big_program print clean
```

2.5.3 Updating Makefiles with make

A makefile can be used to update itself. Example 2-5 shows code from the end of a makefile that uses the compiler to calculate dependencies and write a new makefile. Having code like this at the end of your makefile frees you from keeping track of file dependencies.

Example 2-5: Automatic Makefile Updating

```
depend:
     rm -f eddep makedep
     ${CC} ${CIDIRS} ${DFLAGS} -Em *.c | \
     awk ' { if ($$1 != prev) { print rec; rec = $$0; prev = $$1; } \
          else { if (length(rec $$2) > 78) { print rec; rec = $$0; } \
            else rec = rec " " $$2 } } \
               END { print rec } ' > makedep ; \
     ${ECHO} '/^# DO NOT DELETE THIS LINE/+1,$$d' >eddep
     ${ECHO} '$$r makedep' >>eddep
     ${ECHO} 'w Makefile' >>eddep
     rm -f Makefile.bak
     mv Makefile Makefile.bak
     chmod 755 Makefile.bak
     ex - Makefile.bak < eddep
     ${ECHO} '# DEPENDENCIES MUST END AT END OF FILE' >> Makefile
     ${ECHO} '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> Makefile
     ${ECHO} '# see make depend above' >> Makefile
     rm -f eddep makedep
     chmod 444 Makefile

# DO NOT DELETE THIS LINE -- make depend uses it
```

Issuing the `make` command with `depend` as the target writes a new makefile containing your program's current dependencies. The output is similar to the following:

```
# DO NOT DELETE THIS LINE -- make depend uses it

declarator.o: declarator.c ./../cs_common/master.h
declarator.o: /sybil/ANSI/release/include/setjmp.h
declarator.o: /sybil/ANSI/release/include/sys/types.h
declare.o: declare.c /sybil/ANSI/release/include/string.h
declare.o: ./../cs_common/master.h /sybil/ANSI/release/include/setjmp.h
expression.o: expression.c ./../cs_common/root_stub.h
initial.o: initial.c /sybil/ANSI/release/include/string.h
initial.o: ./../cs_common/master.h /sybil/ANSI/release/include/setjmp.h
lex.o: /sybil/ANSI/release/include/ctype.h ./preproc.h ./../cs/locator.h
lex.o: ./../compiler_message/message_ids.h
macro.o: macro.c /sybil/ANSI/release/include/string.h
macro.o: /sybil/ANSI/release/include/time.h
parse.o: parse.c /sybil/ANSI/release/include/ctype.h
parse.o: /sybil/ANSI/release/include/stdlib.h ./../cs_common/root_stub.h
# DEPENDENCIES MUST END AT END OF FILE
# IF YOU PUT STUFF HERE IT WILL GO AWAY
# see make depend above
```

Debugging Programs 3

The `dbx` debugger is an interactive, symbolic debugger that you can use to find errors in your program code. You can use `dbx` to debug a running program or to examine a core file. The `dbx` debugger can perform the following tasks:

- Display source code with line numbers
- Execute source code conditionally
- Execute source code one line at a time
- Execute source code one machine instruction at a time
- Set and remove breakpoints
- Trace a line, a variable, or a routine
- Trap signals sent to your program
- Call routines outside of normal program flow
- Display the contents of variables
- Assign a value to a variable
- Debug the ULTRIX kernel, `/vmunix`. (For information about debugging the ULTRIX kernel, see Appendix D.)

ULTRIX also provides a DECwindows interface to `dbx`, which is called `dxdb`. For information on `dxdb`, see the *Guide to the dxdb Debugger* in the ULTRIX Worksystem Software documentation.

To demonstrate the `dbx` utility, this chapter gives an example of debugging a sample program. Although this example works on both RISC and VAX systems, on VAX systems, the output may differ from the example. In particular, line numbers and event numbers might be different. For information about the sample program and a listing of the entire program, see Section 3.4.

3.1 The dbx Command

To use `dbx`, you must first compile your program using the `-g` option on the compiler command line. This option provides symbol table information needed by `dbx`. (For information about the compiler commands and the `-g` option, see the reference page that describes the compiler you use.)

To invoke `dbx` you issue the `dbx` command at the shell prompt. The `dbx` command has the following syntax:

dbx [*options*] [*object*] [*core*]

Table 3-1 summarizes the options available on the `dbx` command line.

Table 3-1: dbx Command Options

Option	Purpose
`-c`	Selects a command file other than `.dbxinit`. (For information on creating a command file, see Section 3.3.)
`-i`	Invokes `dbx` in interactive mode. This option causes the debugger to not treat source lines beginning with number signs (#) as comments.
`-I` *directory*	Adds the directory you name to the list of directories `dbx` uses when it searches for a source file. By default, `dbx` searches the current directory and the directory where *object* is located. You can specify multiple directories by using multiple `-I` options.
`-k`	Maps memory addresses. This option is useful for kernel debugging.
`-r`	Executes the object file you name on the command line immediately. If program execution terminates with an error, `dbx` displays the message that describes the error. You can then either invoke the debugger or let the program continue exiting. The `dbx` debugger reads from `/dev/tty` when you specify the `-r` option and standard input is not a terminal. On RISC systems, if the program executes successfully, `dbx` prompts you for input. On VAX systems, if the program executes successfully, `dbx` exits.
	RISC Specific
`-pixie`	Reads in output from the `pixie` utility. The `pixie` utility is a code profiler. For this option to work, you must have executable `pixie` output and the `nonpixie` executable file in the same directory. The `pixie` output must be named `filename.pixie`, where `filename` is the name of the executable file.

On the command line, the *object* argument names the object file that `dbx` reads as input. Name the object file of the program you want to debug using `dbx`. If you omit the *object* argument, `dbx` prompts you for the name of an object file, as shown:

```
enter object file name (default is 'a.out'):
```

You can either enter the name of an object file or press the Return key. If you press the Return key, `dbx` attempts to read a file named `a.out` from the current directory. If no `a.out` file exists, `dbx` exits.

The *core* argument on the `dbx` command line names a core dump file. When you name a core dump file on the `dbx` command line, you can get information about the state of your program when it failed. The core dump file contains an image of memory at the time your program failed. Using `dbx` commands, you can display the value of variables at the time of failure, the values in registers, and so on. The debugger displays information from the core dump file, rather than from memory as it usually does.

If your program takes arguments, do not enter them on the `dbx` command line; enter them as arguments to the `run` command, as described in Section 3.2.3.

3.2 Sample dbx Session

The dbx debugger provides many commands for debugging your program. The following sections take you through an example that demonstrates commonly used commands. The sections also describe the commands and explain the output from the commands.

For information on all the dbx commands, including their syntax, see dbx(1) in the *ULTRIX Reference Pages.*

The sample program is provided on your ULTRIX systems in the file /usr/examples/dbx/dbx_sample.c. To follow the example, first copy the sample program to your area by issuing the following command:

```
% cp /usr/examples/dbx/dbx_sample.c dbx_sample.c
```

Then, compile the sample program by issuing this command:

```
% cc -g dbx_sample.c
```

Finally, invoke dbx, as shown:

```
% dbx
enter object file name (default is 'a.out'): Return
dbx version 2.10
Type 'help' for help.
reading symbolic information ...
main:  28  signal(SIGINT,handler);
(dbx)
```

To exit from dbx, issue the quit command described in Section 3.2.18.

Example 3-2 shows the dbx_sample.c program, which is used throughout the chapter to demonstrate the dbx commands.

3.2.1 Displaying Source Lines (list)

Use the list command to display lines in your program. If you issue the list command without arguments, dbx lists lines beginning at the current source line. You can specify a range of line numbers to list or a beginning line number and the number of lines you want dbx to list.

If you do not specify how many lines to list, dbx lists a default number of lines. How many lines the debugger lists depends on the type of terminal you are using. You can change the default by modifying the $listwindow debugger variable. (For information on modifying debugger variables, see Section 3.3.)

At certain times during program execution, source code is not available to the debugger. For example, suppose you call a C library function and program execution stops while that function is executing. If you issue the list command without arguments, dbx displays the following message:

```
Source not available.
```

The debugger attempts to list source lines beginning at the current line, but it cannot list lines in libc. To list source code lines, you must specify lines in your program to list or name a routine in your program.

The following example demonstrates using `list`:

```
(dbx) list 1,5[1]
     1  /* This program is a small editor that can make very simple
     2   * changes to lines of text.
     3   */
     4
     5  #include <stdio.h>
(dbx) list stredit[2]
    78  void stredit(source)
    79  char source[];
    80  {
    81    register char *start;
    82
    83
    84    if(*source == '\0')
    85         return;
    86    switch(choice) {
    87    /* Convert to upper case */
    88    case 1:
    89         while(*source != '\0'){
    90              if(!isspace(*source))
(dbx)list[3]
    91                   *source = toupper(*source);
    92              source++;
    93         }
    94         break;
    95    /* Convert to lower case */
    96    case 2:
    97         while(*source != '\0'){
    98              if(!isspace(*source))
    99                   *source = tolower(*source);
   100              source++;
   101         }
   102         break;
```

[1] The `list` command displays lines 1 through 5.

[2] The `list` command displays lines in the `stredit()` function, which is one of the functions in the sample program.

[3] The `list` command displays lines beginning at the current line, which is the one following the last line displayed by the previous list command.

3.2.2 Creating Breakpoints (stop)

Use the `stop` command to create a breakpoint in your program. The `stop` command allows you to specify a line number to stop at or a routine to stop in. You can also specify that the debugger stop execution when a variable changes value.

The debugger assigns an event number to each breakpoint. You use the event number to remove the breakpoint. For information about removing breakpoints, see Section 3.2.6.

The following example demonstrates using `stop`:

```
(dbx) stop at 58[1]
[2] stop at "sample.c":58
(dbx) stop in getline[2]
[3] stop in getline
(dbx) stop choice[3]
[4] stop ifchanged choice
```

[1] The `stop` command sets a breakpoint at line 58 in the module named `sample.c`. The program will stop immediately before the debugger executes source code line 58.

The debugger assigns event number 2 to this breakpoint. On RISC systems, the `dbx` debugger uses event number 1 internally, so the first event you create is always number 2.

[2] The `stop` command sets a breakpoint in the `getline()` function. (The `getline()` function is one of the functions in the sample program.) The program will stop immediately before the debugger executes the first source in `getline()`.

The debugger assigns event number 3 to this breakpoint.

[3] The `stop` command will cause program execution to stop when the choice variable changes value.

The debugger assigns event number 4 to this breakpoint.

3.2.3 Running Your Program (run and rerun)

Use the `run` or `rerun` command to run your program under control of the debugger. These commands begin program execution at the beginning of your program. You can pass arguments to your program on the `run` or `rerun` command line. If you specify arguments with the `run` command, issuing `rerun` without arguments passes the same arguments to the program. Otherwise, `run` and `rerun` are identical.

Because the `run` and `rerun` commands always begin execution at the beginning of the program, you cannot use them to continue execution from a breakpoint. For information on continuing execution from a breakpoint, see Section 6.4.

The breakpoints you set always remain active, no matter how many times you reissue the `run` or `rerun` command. For information on deleting breakpoints see Section 6.5.

The following example demonstrates using `run` and `rerun`:

```
(dbx) run [1]



[3]    [getline:69 ,0x400318]     for(i=0; i<MAX ; i++) [2]
(dbx) run "test" [3]


Choose an editing change:

    1 UPPERCASE
    2 lowercase
    3 Initial Capital On All Words
    4 No blanks
    5 Exit

[2]   [main:58 ,0x4002b8]      printf("Enter your choice: ");
(dbx) rerun [4]


Choose an editing change:

    1 UPPERCASE
    2 lowercase
    3 Initial Capital On All Words
    4 No blanks
    5 Exit

[2]   [main:58 ,0x4002b8]       printf("Enter your choice: ");
(dbx)
```

1 The run command executes the program from its beginning. The breakpoint in the getline() function stops program execution.

On VAX systems, the breakpoint on the choice variable stops program execution. The debugger executes that breakpoint when the choice variable is initialized. Issue the cont command to continue program execution until the debugger executes the breakpoint in getline().

2 When a breakpoint stops program execution, dbx displays a message. The message indicates the event number of the breakpoint, the routine and line number in which the stop occurred, the current address of program execution, and the text of the next source line dbx will execute.

3 The run command executes the program from the beginning and passes a string to the program. Because the run command argument supplies input to the program, the program does not execute the getline() function. The breakpoint in getline(), therefore, cannot stop program execution. The breakpoint at line 58 stops program execution.

4 The rerun command executes the program from the beginning. The command passes the same argument as the run command passed and program execution is identical to the execution caused by the run command with the test argument. The breakpoint at line 58 stops the program.

3.2.4 Setting Environment Variables (setenv)

To set an environment variable, issue the setenv command. You can use this command to set the value of an existing environment variable or create a new environment variable. The environment variable is used by both dbx and the program you are running under dbx control.

The following demonstrates using setenv:

```
(dbx) setenv TEXT "test" 1
(dbx) run 2


Choose an editing change:

    1 UPPERCASE
    2 lowercase
    3 Initial Capital On All Words
    4 No blanks
    5 Exit

[2]   [main:58 ,0x4002b8]       printf("Enter your choice: ");
(dbx) setenv TEXT "" 3
(dbx) run 4
[3]   [getline:69 ,0x400318]   for(i=0; i<MAX ; i++)
```

1 The setenv command sets the environment variable TEXT to the value ‘‘test’’.

[2] The `run` command executes the program from the beginning. The program reads input from the environment variable TEXT if it is defined. In this case, the environment variable is defined, so the program does not need to execute the `getline()` function to get input. Program execution stops at the breakpoint at line 58.

[3] The `setenv` command sets the environment variable TEXT to null.

[4] The `run` command executes the program. Because the TEXT environment variable contains a null value, the program must get input. The program executes the `getline()` function and stops at the breakpoint in that function.

3.2.5 Displaying the Status of Debugger Events (status)

Use the `status` command to display the list of active breakpoints and trace events. The `status` command displays the event number and type of each breakpoint and trace event.

On RISC systems, you can create a record event. The `status` command on RISC systems also shows the status of any record events.

On VAX systems, you can redirect the output of the `status` command to a file. Use the right angle bracket (>) as you do at the shell prompt to redirect output.

The following example demonstrates using `status`:

```
(dbx) status [1]
[2] stop at "sample.c":58
[3] stop in getline
[4] stop ifchanged choice
```

[1] The `status` command displays the current debugger events.

3.2.6 Removing Debugger Events (delete)

Use the `delete` command to remove a debugger event, such as a breakpoint or trace event. To remove all debugger events, issue either the `delete all` or `delete *` command.

Specifying an event number that does not correspond to an existing event has no effect.

When you delete an event, the debugger does not reuse the event number assigned to that event. The debugger always increments the event number by one when it creates a new event.

The following example demonstrates `delete`:

```
(dbx) status [1]
[2] stop at "sample.c":58
[3] stop in getline
[4] stop ifchanged choice
(dbx) delete 4 [2]
(dbx) status [3]
[2] stop at "sample.c":58
[3] stop in getline
(dbx) delete * [4]
```

[1] The `status` command displays the current debugger events.

2 The `delete` command removes debugger event number 4. Event number 4 corresponds to the breakpoint that is set on the choice variable.

3 The `status` command displays the debugger events that are current after `dbx` executes the `delete` command.

4 The `delete` command removes all the breakpoints.

3.2.7 Continuing Execution After a Breakpoint (cont)

Use the `cont` command to continue execution after a breakpoint. When you issue the `cont` command without arguments, one of the following occurs:

- Your program runs until a breakpoint stops execution.
- Your program stops execution due to an error.
- Your program executes successfully to its end.

You can specify a signal name to cause your program to execute as if it received that signal.

On RISC systems, you can specify that execution continue only until a particular line is reached or until a particular routine is reached.

You cannot use the `cont` command to begin program execution. Only the run and rerun commands can begin program execution. For information on those commands, see Section 3.2.3.

The following example demonstrates using `cont`:

```
(dbx) stop at 48 1
[5] stop at "sample.c":48
(dbx) stop at 51 2
[6] stop at "sample.c":51
(dbx) run 3


[5] stopped at   [main:48 ,0x400254]      getline(str);
(dbx) cont 4
Enter a text line: test
[6] stopped at   [main:51 ,0x400268]      printf("\n");
```

1 The `stop` command sets a breakpoint at line 48.

2 The `stop` command sets a breakpoint at line 51

3 The `run` command executes the program from the beginning. Program execution stops when `dbx` executes the breakpoint at line 48.

4 The `cont` command continues execution from where it stopped.

5 The program prompts for input. Enter ''test''. The program continues to execute until `dbx` executes the breakpoint at line 51.

3.2.8 Executing One Source Line at a Time (next and step)

Use the `next` command to execute the next source line. If the source line contains a call to a routine, `dbx` executes the entire routine. Program execution stops after the called routine returns to the calling routine.

On RISC systems, you can specify an integer that determines the number of times the debugger executes the `next` command.

Use the `step` command to execute the next source line. If the source line contains a call to a routine, the `dbx` debugger stops at the first line of the routine.

On RISC systems, you can specify an integer that determines the number of times the debugger executes the `step` command.

To execute your source program one line at a time from the beginning, set a breakpoint at line 1 in the program. Then, issue the `run` command. After program execution stops at the breakpoint, use `next` or `step` to continue program execution.

When you are using the `next` and `step` commands, breakpoints appear to be inactive. These commands stop your program at the end of each source line. The debugger does not issue a message for other breakpoints or trace events that have been reached. The exception to this rule is when you are using the `next` command and the debugger encounters a breakpoint in a routine. In this case, the breakpoint in the routine operates normally.

The following example demonstrates using `next` and `step`:

```
(dbx) run 1


[6] stopped at   [main:48 ,0x400254]     getline(str); 2
(dbx) step 3
[getline:69 ,0x400318]     for(i=0; i<MAX ; i++)
(dbx) step 4
[getline:71 ,0x400330]     st[i]=getchar();
(dbx) step 5
Enter a text line: test 6
  [getline:72 ,0x4003b0]       if (st[i]=='\n')
(dbx) run 7


[6] stopped at   [main:48 ,0x400254]     getline(str);
(dbx) next 8
Enter a text line: test
[main:50 ,0x40025c] i = strlen(str);
(dbx) next 9
[7]    [main:51 ,0x400268]      printf("\n");
(dbx) status 10
[5] stop at "sample.c":48
[6] stop at "sample.c":51
(dbx) delete 5 11
```

1 The `run` command executes the program from the beginning. The breakpoint at line 48 stops program execution.

2 The message displays the name of the routine in which execution stopped, the line number where execution stopped, the current address of program execution, and the source line that will be executed when the program continues.

3 The `step` command executes the source code at line 48 in the source program. Line 48 in the source program contains the call to the `getline()` function.

4 The `step` command executes line 69 in the source program. Line 69 is the first line in `getline()`.

5 The `step` command executes the source code at line 71 in the source program. Line 71 in the source program is the second executable line in `getline()`. This line reads input; enter ‘‘test’’.

6 The `run` command executes the program from the beginning. The breakpoint at line 48 stops program execution.

7 The `next` command executes line 49 in the source program. That line contains a call to `getline()`. The next command executes the entire `getline()` function. The `getline()` function prompts you for input. Enter "test".

8 The `next` command executes line 50 in the source program. Line 50 is the first line after the return from `getline()` in the source program.

9 The `status` command displays the currently active breakpoints.

10 The `delete` command removes the breakpoint at line 48.

3.2.9 Tracing Program Execution (trace)

Use the `trace` command to trace the execution of the program.

On RISC systems, you can use `trace` to trace the following:

- The execution of a particular source line.
- The call to and return from a routine.
- The value of a variable at a particular source line. You can specify a condition to limit the trace to times when the condition is true.
- The value of a variable in a routine. You can specify a condition to limit the trace to times when the condition is true.

On VAX systems, you can use `trace` to trace the following:

- The execution of a source line in the program. You can specify a condition to limit the trace to times when the condition is true.
- The execution of each source line in a routine or in the entire program. You can specify a condition to limit the trace to times when the condition is true.
- The value of an expression at the specified source line. You can specify a condition to limit the trace to times when the condition is true.
- The value of a variable. You can limit the trace to a particular source line, a routine, or times when a condition you specify is true.

The following example demonstrates using `trace`:

```
(dbx) stop in getline 1
[8] stop in getline
(dbx) run 2


[8] stopped at   [getline:69 ,0x400318]  for(i=0; i<MAX ; i++)
(dbx) trace i in getline 3
[8] trace sample.getline.i in getline
(dbx) cont 4
Enter a text line:test
[8] sample.getline.i changed before [getline: line 71]: 5
            new value = 1;
[8] sample.getline.i changed before [getline: line 71]:
               old value = 1;
               new value = 2;
[8] sample.getline.i changed before [getline: line 71]:
               old value = 2;
```

```
                new value = 3;
[8] sample.getline.i changed before [getline: line 71]:
                old value = 3;
                new value = 4;
[8] sample.getline.i changed before [getline: line 76]:
                old value = 4;
                new value = 5;
[6] stopped at   [main:51 ,0x400268]     printf("\n"); 6
```

1 The `stop` command sets a breakpoint in the `getline()` function.

2 The `run` command executes the program from the beginning. The breakpoint in `getline()` stops program execution.

3 The `trace` command traces the value of the i variable in the scope of `getline()`.

4 The `cont` command continues execution. When the program prompts you for input, enter the string "test".

5 The `trace` command displays a message each time the value of the i variable changes. The message displays the fully qualified name of the variable being traced, the line at which the variable changed value, the old value of the variable (if applicable), and the new value of the variable.

The i variable changes value five times during the execution of `getline()`.

6 Program execution stops when the debugger executes the breakpoint at line 51.

3.2.10 Assigning Values to Program Variables (assign)

Use the `assign` command to assign a value to a program variable. The value you specify must have the same data type as the program variable. For example, you cannot assign a floating point number to an integer variable. You can assign the value of one variable to another variable. Name the variable into which you want to store a value on the left-hand side of the equal sign (=). Name the other variable on the right-hand side of the equal sign.

The following example demonstrates using `assign`:

```
(dbx) assign i = 100E3 1

incompatible types
(dbx) assign i = 22 2
22
```

1 The `assign` command attempts to store a floating point value in the integer variable i. The debugger displays the message "incompatible types," which indicates that it cannot store the floating point value.

On VAX systems you receive the following message when you issue the `assign` command:

```
i not active
```

Issue the `run` command. Program execution stops when the debugger executes the breakpoint in the `getline()` function. At that point, the i variable is active, and you can assign a value to it.

1 The `assign` command stores the value 22 in the i variable. The debugger displays the value to indicate that it has been stored successfully.

3.2.11 Displaying the Value of Variables (print and printf)

Use either the `print` or `printf` command to display the value of a particular variable. The `print` command displays the value of a variable without formatting it. The `printf` command allows you to specify a format for the variable value. You can specify the same formats as you can using the `printf()` C library routine, except that the `dbx` `printf` command does not support the %s format. (For more information about the `printf()` C library routine, see `printf`(3) in the *ULTRIX Reference Pages.*)

You can specify expressions, such as 1 + 2, on the print command line. The debugger resolves the expression and displays the result. For more information about specifying debugger expressions, see `dbx`(1) in the *ULTRIX Reference Pages.*

The debugger encloses strings in quotation marks when it displays them. If the string contains a carriage return character, `dbx` displays the closing quotation mark as the first character in the second line of the display.

The following example demonstrates using the `print` and `printf` commands:

```
(dbx) print i[1]
22
(dbx) printf"%x %d\n",i,i[2]
16 22
(dbx) print str[3]
"test
"
```

[1] The `print` command displays the value of the i variable.

[2] The `printf` command displays the value of the i variable using hexadecimal and decimal notation.

[3] The `print` command displays the value of the str variable.

3.2.12 Displaying the Names of Active Routines (where)

Use the `where` command to display the names of active routines. On RISC systems, you can specify an integer that determines how many stack levels the debugger displays.

The following example demonstrates using `where`:

```
(dbx) run[1]


[7]  [getline:69 ,0x400318]    for(i=0; i<MAX ; i++)
(dbx) where [2]
>  0 getline(st = 0x7fffbe78 = "") ["sample.c":69, 0x400318]
   1 main(argc = 1, argv = 0x7fffbee4) ["sample.c":48, 0x400258]
```

[1] The `run` command executes the program from the beginning. The breakpoint in the `getline()` function stops program execution.

[2] The `where` command displays the following information about the active routines:

- The right angle bracket (>), which indicates the debugger's current scope.
- The activation level of each routine.

- The routine name.
- The contents of any arguments passed to the routine.
- The source module name and line number.
- The program counter for the current point of execution. In this case, the first program counter value (0x400318) is the program counter at the point where execution stopped. The second program counter value (0x400258) is the program counter at the call to the `getline()` function.

3.2.13 Changing the Debugger's Scope (func)

Use the `func` command to change the debugger's scope. By default, the debugger's scope is the active routine. The debugger uses its scope to resolve references to variable names and line numbers.

When you issue the `func` command, you change the debugger's scope to a routine other than the current one.

Changing the debugger's scope to a new routine does not make that routine active. In other words, you cannot use the `func` command to alter program flow.

On RISC systems, you can specify an integer on the `func` command line. The integer specifies the activation level of an active routine. Once you issue the `func` command with an integer, the routine that corresponds to that activation level becomes the debugger's scope.

The following example demonstrates using `func`:

```
(dbx) where [1]
>  0 getline(st = 0x7fffbe78 = "") ["sample.c":69, 0x400318]
   1 main(argc = 1, argv = 0x7fffbee4) ["sample.c":48, 0x400258]
(dbx) func main [2]
main:  48  getline(str);
(dbx) where [3]
   0 getline(st = 0x7fffbe78 = "") ["sample.c":69, 0x400318]
>  1 main(argc = 1, argv = 0x7fffbee4) ["sample.c":48, 0x400258]
(dbx) func stredit [4]
stredit:  84  if(*source == '\0')
(dbx) where [5]
   0 getline(st = 0x7fffbe78 = "") ["sample.c":69, 0x400318]
   1 main(argc = 1, argv = 0x7fffbee4) ["sample.c":48, 0x400258]
```

[1] The `where` command displays the list of active routines. The debugger's scope is the `getline()` function.

[2] The `func` command changes the debugger's scope to `main()`. The message displays the routine name of the new scope, the current source line number in that routine, and the text of the next source line `dbx` will execute.

[3] The `where` command verifies the change in the debugger's scope.

[4] The `func` command changes the debugger's scope to the `stredit()` function.

[5] The `where` command displays the list of active routines. Because `stredit()` is inactive, that routine is not on the list. Therefore, the debugger does not display a pointer to the routine that contains the current scope.

3.2.14 Displaying Fully Qualified Variable Names (which and whereis)

Use either the `which` or `whereis` command to display the fully qualified name of a variable. The `which` command displays the fully qualified name of the variable you name, as defined by the current debugger scope. The `whereis` command displays the fully qualified name of each occurrence of the specified variable.

The following example demonstrates using `which` and `whereis`:

```
(dbx) which i [1]
sample.getline.i
(dbx) whereis i [2]
sample.getline.i sample.main.i
```

[1] The `which` command displays the fully qualified name of the i variable, including the program module name, the routine name, and the variable name. Because the debugger's current scope is the `getline()` function, the fully qualified name is sample.getline.i.

[2] The `whereis` command displays the fully qualified name of all i variables that are declared in the program.

3.2.15 Calling Routines (call)

Use the `call` command to execute a routine in your program. The `call` command executes the routine you name on the command line. You can pass parameters to the routine by specifying them as arguments to the `call` command.

The `call` command does not alter the flow of your program. Once the routine returns, program execution resumes at the point where you issued the `call` command.

The following example demonstrates using `call`:

```
(dbx) stop in stredit [1]
[9] stop in stredit
(dbx) call stredit (&str) [2]
[9]   [stredit:84 ,0x400448]   if(*source == '\0')
(dbx) status [3]
[6] stop at "sample.c":51
[7] stop in getline
[8] trace sample.getline.i in getline
[9] stop in stredit
(dbx) delete 7; delete 8 [4]
```

[1] The `stop` command sets a breakpoint in the `stredit()` function.

[2] The `call` command begins executing the object code associated with `stredit()`. The str argument passes a string by reference to `stredit`.

On VAX systems, you must pass the str argument by value. Issue the following `call` command to call the `stredit()` function:

```
(dbx) call stredit (str)
```

This command passes the type that `stredit()` expects on a VAX system.

[1] The `status` command displays the currently active breakpoints and trace events.

[2] The `delete` commands delete the breakpoint at line 51 and the trace event. When you specify more than one command on the `dbx` command line, you must separate the commands with a semicolon (;).

3.2.16 Catching and Ignoring Signals (catch and ignore)

Use the `catch` and `ignore` commands to determine which signals `dbx` catches.

The debugger catches some signals by default. To see which signals `dbx` is currently catching, issue the `catch` command without arguments.

To cause `dbx` to catch a signal, name that signal on the `catch` command line. When `dbx` catches a signal, it intercepts that signal before it reaches your program.

To cause `dbx` to ignore a signal, issue the `ignore` command. The `ignore` command causes the debugger to pass the named signal to your program, rather than intercepting that signal.

The following example demonstrates using `catch` and `ignore`:

```
(dbx) catch 1
INT QUIT ILL TRAP IOT EMT FPE BUS SEGV SYS PIPE TERM URG STOP TTIN TTOU
IO XCPU XFSZ VTALRM PROF WINCH LOST USR1 USR2
(dbx) ignore int 2
(dbx) run 3


Enter a text line: Ctrl/C 4
^C disabled - Re-enter input:
test 5
[7] stopped at   [main:51 ,0x400268]     printf("\n");
(dbx) catch int 6
(dbx) run 7


Enter a text line: Ctrl/C 8
Interrupt [read.read:18 +0x8,0x403fa8]
       Source not available
```

1 The `catch` command without arguments displays a list of the signals that `dbx` is currently catching. In this case, the list displays the signals that `dbx` catches by default.

2 The `ignore` command causes `dbx` to ignore the SIGINT signal. You can generate the SIGINT signal by pressing Ctrl/C at the keyboard.

3 The `run` command begins program execution from the beginning.

4 When the program prompts for input, enter Ctrl/C. Because `dbx` ignores that signal, the signal is passed through to the program, which contains a routine to handle Ctrl/C. That routine executes and the program continues.

5 The `catch` command specifies that `dbx` catch the SIGINT signal.

6 The `run` command begins program execution

7 When the program prompts for input, enter Ctrl/C again.

8 The debugger interprets the Ctrl/C signal and stops program execution during the call to the `read()` function. The debugger displays a message that lists the following:

- The signal that interrupted program execution
- The routine in which execution stopped
- The current program counter
- The text of the current source line, if possible

In this case, the debugger cannot display the source line from `read()` because that function is in the C library; the debugger cannot display source code from the C library.

3.2.17 Repeating Commands (pressing Return, history, and !)

You can repeat commands in `dbx` by pressing the Return key and using the history list. Pressing the Return key at the `(dbx)` prompt reexecutes the last command you entered.

The `dbx` debugger maintains a list of the commands you issue in its history list. You can repeat commands in the history list using the exclamation point (!).

By default, the debugger stores the previous 20 commands in its history list. You can change the number of commands the debugger stores by assigning a different value to the `$lines` variable on RISC systems or the `$historywindow` variable on VAX systems. (For information on setting the value of debugger variables, see Section 3.3.)

The following example demonstrates repeating commands using the Return key and the history list:

```
(dbx) history 1
197     print str
198     run
199     where
200     where
201     func main
202     where
203     func stredit
204     where
205     which i
206     whereis i
207     stop in stredit
208     call stredit (&str)
209     status
210     delete 7; delete 8
211     catch
212     ignore int
213     run
214     catch int
215     run
216     history
(dbx) !204 2
(!204 = where)
>  0 read.read(0x0, 0x0, 0x0, 0x0, 0x0) ["../read.s":18, 0x403fa8]
   1 _filbuf(0x0, 0x46, 0x7fffbd0e, 0x32, 0x15) ["../filbuf.c":144,
     0x40119c]
   2 getline(st = 0x7fffbe78 = "") ["sample.c":71, 0x400394]
   3 main(argc = 1, argv = 0x7fffbee4) ["sample.c":48, 0x400258]
(dbx) !r 3
(!r = run)


Enter a text line: test
[6] stopped at   [main:51 ,0x400268]     printf("\n");
(dbx) Return 4
run
```

```
Enter a text line: test
[6] stopped at   [main:51 ,0x400268]     printf("\n");
```

1 The `history` command displays the history list, which contains the last 20 commands issued during the session.

2 The `!204` command executes command number 204 in the history list. In this case, command number 204 is the `where` command.

3 The `!r` command executes the most recent command that begins with the letter ''r.'' In this case, `!r` executes the `run` command. When the program prompts for input, enter the string ''test''.

4 Pressing the Return key executes the last command issued; in this case, the `run` command. When the program prompts for input, enter the string ''test''.

3.2.18 Ending a Debugging Session (quit)

To leave `dbx` when you complete a debugging session, issue the `quit` command.

The following example demonstrates using `quit`:

```
(dbx) quit
%
```

The `quit` command ends the `dbx` session and returns you to the shell prompt.

3.3 Initializing dbx

You can create an initialization file for `dbx` that contains commands you normally issue at the beginning of each `dbx` session. You must name the initialization file `.dbxinit`. The debugger searches for the `.dbxinit` file in your current directory. If the debugger finds no `.dbxinit` file in your current directory, it searches your home directory (the directory assigned to the $HOME environment variable). Each time you invoke the debugger, it reads and executes the commands in `.dbxinit`. Example 3-1 contains a sample dbx initialization file.

Example 3-1: Sample dbx Initialization File

```
alias stopget "stop in getline" 1
set $listwindow = 5 2
set $lines = 25 3
setenv EDITOR = ex 4
```

1 The `alias` command defines an alias for the `stop` command. Issuing the stopget alias sets a breakpoint at the `getline()` function.

2 The `set` command changes the value of the `$listwindow` variable to 5. Once the debugger executes this `set` command, the `list` command will display 5 lines by default.

3 The `set` command changes the value of the `$lines` variable to 25. The `$lines` variable controls how many history lines the debugger stores. The debugger will store 25 commands in the history list after it executes this command.

4 The `setenv` command sets the environment variable EDITOR to ex. When this environment variable is set to the `ex` editor, that editor is invoked when you issue the `edit` command in `dbx`.

For more information on debugger variables and their default values, see `dbx(1)` in the *ULTRIX Reference Pages.*

3.4 Sample Program

The sample program used in this chapter is a simple editor that reads a line from standard input, performs some changes, and writes the modified line to standard output. Example 3-2 shows the complete program.

Example 3-2: Sample Editor Program

```
/* This program is a simple editor that can make changes
 * to lines of text.
 */

#include <stdio.h>
#include <signal.h>
#define MAX 80

void getline();
void stredit();
void handler();
extern char *getenv();

int choice = 0;

main(argc,argv)                                         1
int argc;
char **argv;
{
      char str[MAX];
      char *tmp;
      char newline;
      int i;

      /*
       * Declare a signal handler for ^C.
       */
      signal(SIGINT,handler);
      /*
       * A text string argument may be entered:
       * 1. as a command line argument
       * 2. as the value of an environment variable
       * 3. interactively
       *
       * Once a command line argument or environment string has
       * been processed, the user is prompted for additional text.
       * If both a command line argument and an environment string
       * are given, only the command line argument is processed.
       */
      str[0]=' ';
      if(argc > 1)
            strncpy(str,*++argv,MAX);
      else if ((tmp = getenv("TEXT")) != 0)
            strncpy(str,tmp,MAX);

      if(str[0]==' '){
            printf("\n\nEnter a text line: ");
            getline(str);
      }
      i = strlen(str);
      printf("\n");
      printf("Choose an editing change:\n\n");
      printf("    1  UPPERCASE\n");
```

Example 3-2: (continued)

```
      printf("    2  lowercase\n");
      printf("    3  Initial Capital On All Words\n");
      printf("    4  No blanks\n");
      printf("    5  Exit\n\n");
      printf("Enter your choice: ");
      scanf("%d%c", &choice,&newline);
      stredit(str);
      printf("\n%s\n", str);
}

void getline(st)                                        2
char *st;
{
       int i;

       for(i=0; i<MAX ; i++)
       {
           st[i]=getchar();
           if (st[i]=='\n')
               break;
       }
       st[++i]=' ';
}

void stredit(source)                                    3
char source[];
{
      register char *start;


      if(*source == ' ')
            return;
      switch(choice) {

      /* Convert to upper case */
      case 1:
            while(*source != ' '){
                  if(!isspace(*source))
                        *source = toupper(*source);
                  source++;
            }
            break;

        /* Convert to lower case */
      case 2:
            while(*source != ' '){
                  if(!isspace(*source))
                        *source = tolower(*source);
                  source++;
            }
            break;

      /* Capitalize first letter of each word */
      case 3:
            if(!isspace(*source))
                  *source = toupper(*source);
            source++;
            while(*source != ' '){
                  if(isspace(*(source-1)) && !isspace(*source))
                        *source = toupper(*source);
                  source++;
            }
            break;

      /* Remove all blanks */
      case 4:
            start=source;
            while(*source != ' '){
                  while(*source && isspace(*source))
                        source++;
                  while(*source && !isspace(*source))
```

Example 3-2: (continued)

```
                        *start++ = *source++;
            }
            *start = *source;
            break;

        case 5:
            exit(0);

        default:
            strcpy(source,"Invalid edit choice.\n");
            break;
        }
}
/*
 * Signal handler for ^C
 */
void
handler(sig, code, scp)                                    4
int sig, code;
struct sigcontext *scp;
{
        fprintf(stderr,"\n\n^C disabled -  Re-enter input:\n");
}
```

1 The `main()` function calls two other functions, `getline()` and `stredit()`. The `main()` function also displays messages that help the application user give the appropriate input to the program. The user can choose an editing change. The choices are as follows:

- Display the string in all uppercase letters
- Display the string in all lowercase letters
- Display the string with initial capital letters on all words
- Display the string without blanks
- Exit from the program

2 The `getline()` function gets input from the application user.

3 The `stredit()` function performs the editing change. The function is divided into a case statement, one case for each possible editing change. The default case is that the program fails with the message "Invalid edit choice."

4 The signal handler ensures that the program continues after the application user enters Ctrl/C.

3.5 Built-in dbx Command Aliases

You can use the `alias` command to create aliases for `dbx` commands. In addition, the debugger has a set of predefined aliases that you can use. Table 3-2 lists the predefined aliases.

Table 3-2: Predefined dbx Command Aliases

Alias	Command Description
c	Continues program execution after a breakpoint
d	Deletes the specified item from the status list

Table 3-2: (continued)

Alias	Command Description
`e`	Edits the specified file
`j`	Displays the events on the status list
`n`	Executes the next line without stopping in routines
`p`	Displays the value of the specified expression or variable
`q`	Ends the debugging session
`r`	Reruns the program
`s`	Executes the next line, stopping after each line in any routine
`t`	Performs a stack trace
RISC Specific	
`a`	Assigns a value to a program variable
`b`	Sets a breakpoint at a specified line
`bp`	Stops in a specified routine
`f`	Moves to the specified activation level on the stack
`g`	Goes to the specified line and begins executing the program there
`h`	Lists all items currently on the history list
`l`	Lists the next 10 lines of source code
`li`	Lists the next 10 machine instructions
`ni` or `Si`	Executes the specified number of assembly code instructions without stopping in any routine (`ni`) or stopping after each line in any routine (`Si`).
`pd`	Displays the value of the specified expression or variable in decimal notation
`pi`	Replays `dbx` commands that were saved with the `record input` command
`po`	Displays the value of the specified expression or variable in octal
`pr`	Displays the value of each register
`px`	Displays the value for the specified variable or expression in hexadecimal notation
`ri`	Records each command you enter in the specified file
`ro`	Records all debugger output in the specified file
`S`	Executes the specified number of lines stopping at each line in any routine
`si`	Executes the specified number of assembly code instructions
`u`	Lists the previous 10 lines
`w`	Lists the 5 lines preceding and following the current line
`W`	Lists the 10 lines preceding and following the current line
`wi`	Lists the 5 machine instructions preceding and following the current machine instruction

Table 3-2: (continued)

Alias	Command Description
	VAX Specific
`h`	Lists the help text that describes `dbx` commands
`l`	Lists the number of lines specified by the `$listwindow` variable

Checking Programs and Improving Performance 4

This chapter describes several ULTRIX programs that can help programmers check their code for errors before compiling it, and improve program performance once the program works.

4.1 Checking C Source Files with lint

The `lint` command checks C source files for code that is wasteful, nonportable, or likely to cause bugs. Because `lint` is stricter and quicker than most compilers, run source files through `lint` before compiling them. (However, `lint` is superfluous when used with the `c89` compiler, which performs the same kind of checking as `lint.`) The `lint` command writes messages to stdout for every error or questionable usage. For complete information on `lint` and its options, see `lint(1)` in the *ULTRIX Reference Pages.*

The following example shows sample output from `lint`:

```
% lint program.c
program.c:
program.c(73): warning: c unused in function getline  1
printf returns value which is always ignored          2
scanf returns value which is always ignored           3
```

1 The first message, calls attention to line 73:

```
getline (st)
    char *st;
{
    char c;              /* This is line 73 */
    int i;

    for(i=0; i<=MAX ; i++)
    {
        st[i]=getchar();
        if (st[i]=='\n')
            break;
    }
    st[++i]='\0';
}
```

The variable `c`, declared in the function `getline`, is never used and should be deleted.

2 The return value from `printf` is not checked. Checking the return value of every function is a good programming practice.

3 The return value from `scanf` is not checked. Checking the return value of every function is a good programming practice.

4.2 Monitoring Program Execution with ctrace

The `ctrace` command allows you to watch a C program's flow and observe changes to variables, looking for unexpected behavior. Running `ctrace` on a source file places additional code into the file; this code causes executable statements and referenced or modified variables and their values to be written to stdout during the program's execution. Your C source file must compile without errors before you use `ctrace` on it. For more information, see `ctrace(1)` in the *ULTRIX Reference Pages*.

To use `ctrace`, follow these steps:

1. Run `ctrace` on a C source file. The `ctrace` command sends its output (the modified file) to stdout, so redirect stdout to a file; for example:

   ```
   %  ctrace  program.c  >  ctrace.c
   ```

2. Compile and link the expanded code; for example:

   ```
   %  cc  ctrace.c
   ```

3. Run the program; for example:

   ```
   %  a.out
   ```

As the program runs, its source lines are written to stdout, as shown in Example 4-1.

Example 4-1: Sample ctrace Output

```
25 [1]     for (j = 0 ; j < 10 ; j++)
           /* j == 0 */
26             rec_1.buf[j] = i + 1;
               /* j == 0 */
          [2]  /* i == 0 */
               /* rec_1.buf[j] == 1 */
25         for (j = 0 ; j < 10 ; j++)
           /* j == 1 */
26             rec_1.buf[j] = i + 1;
               /* j == 1 */
               /* i == 0 */
               /* rec_1.buf[j] == 1 */
[3] /* repeating */
    /* repeated 8 times */
25         for (j = 0 ; j < 10 ; j++)
           /* j == 10 or '\n' */
```

[1] The numbers on the left are line numbers relative to the original C source file.

[2] Lines that look like C comments display information about the preceding line.

[3] Loops are detected by `ctrace`, which displays the looping code only once but tells how many repetitions occur.

4.2.1 Tracing Only Certain Functions

Sifting through the trace of a large program is tedious. A bug can often be isolated to certain functions, or certain functions can be dismissed as the source of a problem. To discriminate among functions, use the `-f` and `-v` options to `ctrace`:

-f *functions*	Trace only these functions

-v *functions*	Trace all functions except these

For example:

```
% ctrace -f getline main program.c > ctraced.c
```

The preceding command creates the file `ctraced.c`, which (when compiled, linked, and run) shows a trace of the functions getline( ) and main( ). The following command produces the file `ctraced.c`, which (when compiled and run) shows a trace of the entire program *except* the functions getline( ) and main( ):

```
% ctrace -v getline main crude_editor.c > ctraced.c
```

4.2.2 Tracing Only Certain Sections of Code

To trace only certain sections of code, insert the ctroff( ) and ctron( ) functions around code you do not want to trace. The ctroff( ) and ctron( ) functions turn `ctrace` off and on, respectively, as shown in Example 4-2.

Example 4-2: Tracing Certain Sections with ctrace

```
#include <stdio.h>

main()
{
    FILE *fp;
    struct rec
           { char type;
             short buf[10];
           } rec_1;
    int i, j, status;

    fp = fopen("data_file.txt", "w+");

ctroff();  /****  Turn off tracing  ****/

    for (i = 0 ; i < 3 ; i++)
    {
        for (j = 0 ; j < 10 ; j++)
            rec_1.buf[j] = i + 1;

        rec_1.type = 0x31 + i;
        fwrite(&rec_1, sizeof(rec_1), 1, fp);
    }

ctron();  /****  Turn tracing back on  ****/

    fseek(fp, sizeof(rec_1), 0);

    fread(&rec_1, sizeof(rec_1), 1, fp);

    printf("The second structure:\n\tType:\t%c\n\tContents: ",
           rec_1.type);

    for(i = 0 ; i < 10 ; i++)
        printf(i != 9 ? "%d " : "%d\n", rec_1.buf[i]);

    fclose(fp);
    exit(0);
}
```

When run through `ctrace`, compiled, linked, and run, the preceding program produces the following output:

```
 3 main()
12     fp = fopen("data_file.txt", "w+");
       /* fp == 32772 */
14 ctroff();
   /* trace off */
   /* trace on */
27     fseek(fp, sizeof(rec_1), 0);
       /* fp == 32772 */
29     fread(&rec_1, sizeof(rec_1), 1, fp);
       /* fp == 32772 */
31     printf("The second structure:\n\tType:\t%c\n\tContents: ",
              rec_1.type);
       /* rec_1.type == 50 or '2' */ The second structure:
    Type:2
    Contents:
33     for(i = 0 ; i < 10 ; i++)
       /* i == 0 */
34         printf(i != 9 ? "%d " : "%d\n", rec_1.buf[i]);
           /* i == 0 */
           /* rec_1.buf[i] == 2 */ 2
33     for(i = 0 ; i < 10 ; i++)
       /* i == 1 */
34         printf(i != 9 ? "%d " : "%d\n", rec_1.buf[i]);
           /* i == 1 */
           /* rec_1.buf[i] == 2 */ 2
   /* repeating */ 2 2 2 2 2 2 2 2
   /* repeated 8 times */
33     for(i = 0 ; i < 10 ; i++)
       /* i == 10 or '\n' */
36     fclose(fp);
       /* fp == 32772 */
37     exit(0);
```

Note that the code between the ctroff( ) and ctron( ) function calls still executes, but the code itself does not appear.

4.3 Profiling Code on RISC Systems

Code profiling shows you where most of your code's execution time is spent. Knowing which sections of code are used most allows you to improve efficiency where it will do the most good. There are three types of code profiling:

- Basic block counting, which counts the number of times each basic block is executed. (A basic block is an instruction sequence entered only at its beginning, and left only at its end.) Basic block counting shows which lines of code are used most.
- Invocation counting, which counts the number of times each routine is invoked.
- PC sampling, which reveals the amount of time spent in various parts of the program by periodically examining the PC (program counter) during the program's execution.

A source code compiler and two other programs, `pixie` and `prof`, are the tools that provide this information. The following sections provide an overview and examples of `pixie` and `prof` usage. For more information, see `pixie`(1) and `prof`(2) in the *ULTRIX Reference Pages.*

4.3.1 Basic Block and Invocation Counting

To get basic block and invocation counts for a program, follow these steps:

1. Compile and link, without the `-p` option; for example:

   ```
   %  cc  -o  program  program.c
   ```

2. Run the `pixie` program on the executable file; for example:

   ```
   %  pixie  program
   ```

   The output from `pixie` is two files. One is an equivalent program called `program.pixie` by default (where `program` is the input file name), containing additional code that counts block execution. The other is a file called `program.Addrs` (where `program` is the input file name), which is used by `prof`.

3. Run the `pixie`-modified program, which creates the file `program.Counts` (where `program` is the input file name), which is used by `prof`; for example:

   ```
   %  program.pixie
   ```

4. Run `prof` with the `-pixie` option, which makes the information in `program.Counts` and `program.Addrs` readable and writes it to stdout; for example:

   ```
   prof  -pixie  program
   ```

The basic block counting information appears as shown in Example 4-3.

Example 4-3: Basic Block and Invocation Count Output from prof

```
Profile listing generated Wed Oct 17 17:32:30 1990 with:
1  prof -pixie program

blkclr and bzero (../bzero.s) synonymous: using latter
----------------------------------------------------------------------
* -p[rocedures] using basic-block counts;                            *
* sorted in descending order by the number of cycles executed in     *
* each procedure; unexecuted procedures are excluded                 *
----------------------------------------------------------------------
2
10373 cycles

    cycles %cycles  cum %      cycles  bytes procedure (file)
                                /call  /line

      3481   33.56  33.56          56     14 _flsbuf (../flsbuf.c)
      2653   25.58  59.13         242     18 _doprnt (../doprnt.c)
       818    7.89  67.02         205     13 morecore (../malloc.c)
       652    6.29  73.31         652     21 main (program.c)
       619    5.97  79.27         619     19 _fwalk (../data.c)
       304    2.93  82.20          76     10 malloc (../malloc.c)
    .
    .
    .
----------------------------------------------------------------------
*  -p[rocedures] using invocation counts;                            *
*  sorted in descending order by number of calls per procedure;      *
*  unexecuted procedures are excluded                                *
----------------------------------------------------------------------
3
151 invocations total

     calls  %calls    cum%    bytes procedure (file)

        63   41.72   41.72      608 _flsbuf (../flsbuf.c)
        11    7.28   49.01     4872 _doprnt (../doprnt.c)
        11    7.28   56.29       96 printf (../printf.c)
         6    3.97   60.26       48 sbrk (../sbrk.s)
         4    2.65   62.91       32 close (../close.s)
         4    2.65   65.56      388 malloc (../malloc.c)
    .
    .
    .
----------------------------------------------------------------------
*  -h[eavy] using basic-block counts;                                *
*  sorted in descending order by the number of cycles executed in    *
*  each line; unexecuted lines are excluded                          *
----------------------------------------------------------------------
4
procedure (file)                   line bytes     cycles      %  cum %

_doprnt (../doprnt.c)              305    84        561   5.41   5.41
_flsbuf (../flsbuf.c)              135    52        511   4.93  10.33
_flsbuf (../flsbuf.c)              166    32        441   4.25  14.59
_flsbuf (../flsbuf.c)              131    28        434   4.18  18.77
main (program.c)                    26    48        360   3.47  22.24
_fwalk (../data.c)                  88    20        320   3.08  25.33
    .
    .
    .
```

1 This is the command line that invoked `prof`. When no options that produce a listing or another file are specified, `-procedures` and `-heavy` are used by default.

2 This section shows where the cycles were spent. Each routine in the program is listed on the right. The heading `cum%` stands for cumulative percent; this column lists the total percentage of cycles consumed for the procedure to the right and every procedure above. For example, 67.02% of all cycles were spent on the `morecore`, `_doprnt`, and `_flsbuf` procedures.

3 This section shows how often each procedure was invoked.

4 This section shows how many cycles each program line consumed. The number in the `line` column refers to the line number of the file shown in the left column; for example, the line that consumed the most (561) cycles, was line 305 in the file `doprnt.c`.

Separate runs of a program can produce different basic block information, especially if different input is supplied. However, an average of the different runs can be obtained using the following steps:

1. Run the `pixie`-created program (`program.pixie`, see Section 4.3.1) several times with different input. Each run creates another `program.Counts` file. Between runs, rename `program.Counts` so that it is not overwritten by the next run of `program.pixie`; for example:

   ```
   % program.pixie < input_1
   % mv program.Counts program1.Counts
   % program.pixie < input_2
   % mv program.Counts program2.Counts
   % program.pixie < input_3
   % mv program.Counts program3.Counts
   ```

2. Create a report for the average of all runs, as follows:

   ```
   % prof -pixie program program[123].Counts
   ```

4.3.2 PC Sampling

To get PC (program counter) information for a program, follow these steps:

1. Compile and link the program with the `-p` option; for example:

   ```
   % cc -p program.c -o program
   ```

2. Run the profiled program; for example:

   ```
   % program
   ```

   Profiling data is stored in the profile data file, which has the default name `mon.out`.

3. Run `prof`, which converts the information in the profile data file to a readable form; for example:

   ```
   % prof -procedure program
   ```

The `prof` output for PC sampling appears as shown in Example 4-4.

Example 4-4: PC Sampling Output from prof

```
Profile listing generated Thu Oct 18 16:50:26 1990 with:
   prof -procedure program

blkclr and bzero (../bzero.s) synonymous: using latter
----------------------------------------------------------------------
*  -p[rocedures] using pc-sampling;                                  *
*  sorted in descending order by total time spent in each            *
*  procedure; unexecuted procedures excluded                         *
----------------------------------------------------------------------

Each sample covers 8.00 byte(s) for 25% of 0.0400 seconds

%time     seconds  cum %   cum sec  procedure (file)

 75.0      0.0300   75.0      0.03 open (../open.s)
 25.0      0.0100  100.0      0.04 write (../write.s)
```

You can run the profiled program several times with different inputs to create different profile data files and average the results using `prof`. To create several profile data files, set the PROFDIR environment variable as follows:

- In the C shell:

  ```
  %  setenv  PROFDIR  string
  ```

- In the Bourne shell:

  ```
  %  PROFDIR = string ; export PROFDIR
  ```

Setting PROFDIR causes each profile data file to be saved in a file named `string/pid.program`, rather than `mon.out`. The file name `string/pid.program` is composed of `program`, which is the program's name as it appears in `argv[0]`, and `pid`, which is the process ID of the individual program run and is different for every run. To get the PC sampling average for all runs issue the `prof` command; for example:

```
%  prof  -procedure  program  /string/*.program
```

4.4 Profiling Code on VAX Systems

Code profiling shows you where most of your code's execution time is spent. Knowing which sections of code are used most allows you to improve efficiency where it will do the most good. There are two types of code profiling:

- The flat profile, which shows the following for each routine:
  - The time it used
  - The percentage of total program time it used
  - The number of times it was called
- The call graph profile, which shows everything the flat profile does, plus the following:
  - All parents (routines that called the routine)

- All children (routines called by the routine)
- The percentage of total program time used by the routine and its children
- Other pieces of information, explained in the output file.

A language compiler with a `-pg` option, and the `gprof` program are the tools that provide this information.

4.4.1 Getting a Profile Output File

To obtain a profile output file for a program, follow these steps:

1. Compile and link the program with the `-pg` option; for example:

   ```
   % cc -pg program.c -o program
   ```

2. Run the program, which creates the file `gmon.out`, in which profile information is stored; for example:

   ```
   % program
   ```

3. Run `gprof`. By default, `gprof` searches for `a.out` as the executable file, and `gmon.out` as the profile data file. Output is written to stdout; for example:

   ```
   % gprof program > program.gprof
   ```

In the preceding example, the profile output is written to the file `program.gprof`.

The profile output contains the flat profile followed by the call graph profile. Both profiles are preceded by explanations of the headings and information.

For more information, see `gprof(1)` in the *ULTRIX Reference Pages.*

4.5 Optimizing Programs on a RISC System

When you optimize your code, your program runs faster and its object is smaller in size. Optimizing your program can also speed up development time. For example, your coding time can be reduced if you let an optimizing tool relate programming details to execution time efficiency. This time savings lets you focus on the more crucial global structure of your program. Moreover, programs often yield code sequences that can be optimized regardless of how well you write your source program.

On ULTRIX systems running on the RISC architecture, the optimizer is `uopt`. The `uopt` optimizer is invoked when you use the `-O`, `-O2`, or `-O3` option on the compiler command line. (Note that the DEC Fortran product does not use the `uopt` optimizier. See your DEC Fortran documentation for information on optimizing DEC Fortran programs.) If you omit these options from the compiler command line, limited optimizations are performed. The code generator and assembler phases of the compiler perform the limited optimizations. (For information about using the `-O3` option, see Section 4.5.5.)

4.5.1 Overview of the uopt Optimizer

The `uopt` optimizer improves the performance of object programs by transforming existing code into more efficient coding sequences. Although the same optimizer processes all compiler optimizations, it does distinguish between the various languages supported by the compiler programs to take advantage of the different language semantics involved.

Most compilers perform certain code optimizations, although the extent to which they perform these optimizations varies widely. The RISC compilers perform extensive optimizations compared with the average compiler available. These advanced optimizations are the results of the latest research into better and more powerful compiler techniques.

The compilers perform both machine-independent and machine-dependent optimizations. Machines with RISC architectures provide a better target for machine-dependent optimizations, because the low-level instructions of RISC machines provide more optimization opportunities than the high-level instructions in other machines.

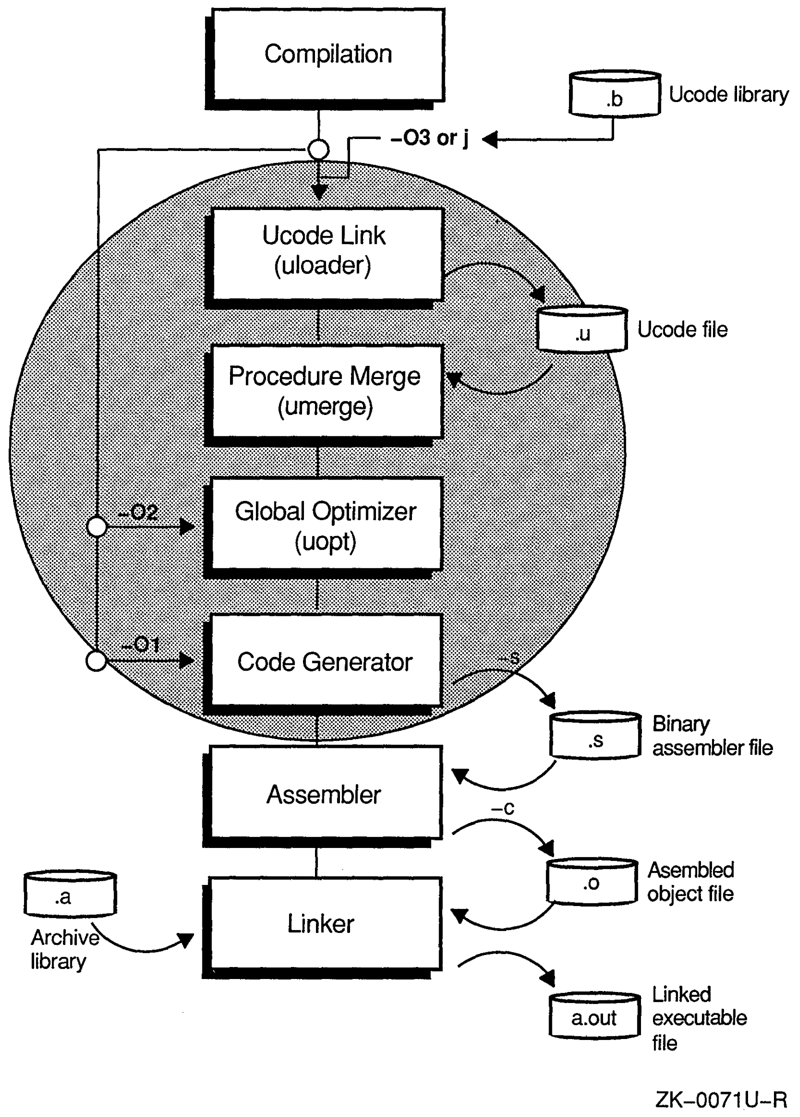
Even optimizations that are machine independent have been found to be effective on machines with RISC architectures. Although most of the optimizations performed by the `uopt` optimizer are machine independent, they have been specifically tailored to this RISC environment.

The RISC architecture emphasizes the use of registers. Therefore, register use has significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from storage. The `uopt` optimizer makes the best possible use of registers.

In allocating registers, the optimizer selects those data items most suited for registers, taking into account their frequency of use and their location in the program structure. In addition, the optimizer assigns values to registers so that their contents move minimally within loops and during procedure invocations.

Figure 4-1 shows the optimization phases of the compiler.

Figure 4-1: Optimization Phases of the Compiler



As the figure shows, the `uload` and `umerge` phases of the compilation permit global optimization among separate units in the same compilation. Often, programs are divided into separate files, called modules or compilation units, which are compiled separately. This saves compile time during program development because a change requires recompilation of only one module rather than the entire program.

Traditionally, program modularity restricted the optimization of code to a single module at a time rather than over the full breadth of the program. For example, calls to procedures that reside in other modules could not be fully optimized together with the code that called them.

The uld and umerge phases of the compiler overcome this deficiency. The uld phase links many modules into a single compilation unit. Then, umerge orders the procedures for optimal processing by uopt.

4.5.2 Things to Consider Before You Optimize a Program

Before you optimize your program, be sure it is fully developed and is relatively error free. Although the optimizer does not alter the flow of control within a program, it may move operations so that the object code does not correspond to the source code.

Once you optimize a program, using dbx to find errors is more difficult. The symbol table that the compiler creates to support symbolic debugging cannot reflect the optimizations that uopt performs. This situation can make it difficult for you to use dbx because, for example, a variable value that dbx displays may not reflect the actual value stored in the variable.

If you are writing Pascal programs, be aware that the -C option of the Pascal compiler inhibits some optimizations. The -C option performs bounds checking. Unless bounds checking is crucial, do not specify the -C option when you compile a program you want to optimize.

Optimizations are most useful in program areas that contain loops. The optimizer moves loop-invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop-invariant code, loops often contain loop-induction expressions that can be replaced with simple increments. In programs composed of mostly loops, global optimization can often reduce the running time significantly.

The following examples illustrate the results of loop optimization on source code that is compiled both with and without the -O compiler option. Example 4-5 shows the source code that contains a loop.

Example 4-5: Source Code of a Program to be Optimized

```
void
left(a, distance)
  char a[];
  int distance;
  {
  int j, length;
  length = strlen(a) - distance;
  for (j = 0; j < length; j++)
    a[j] = a[j + distance];
  }
```

Example 4-6 shows the assembler code that would be output if this program were compiled without the -O option.

Example 4-6: Unoptimized Code Output

```
#    8          for (j=0; j<length; j++)
        sw      $0, 36($sp)     # j = 0
        ble     $24, 0, $33     # length >= j
$32:
 #   9          a[j] = a[j+distance];
        lw      $25, 36($sp)    # j
        lw      $8, 44($sp)     # distance
        addu    $9, $25, $8     # j+distance
        lw      $10, 40($sp)    # address of a
        addu    $11, $10, $9    # address of a[j+
        lb      $12, 0($11)     # a[j+distance]
        addu    $13, $10, $25   # address of a[j]
        sb      $12, 0($13)     # a[j]
        lw      $14, 36($sp)    # j
        addu    $15, $14, 1     # j+1
        sw      $15, 36($sp)    # j++
        lw      $3, 32($sp)     # length
        blt     $15, $3, $32    # j < length
$33:
```

Example 4-7 shows the assembler code that would be output if this program were compiled with the `-O` option.

Example 4-7: Optimized Code Output

```
 #   8          for (j=0; j<length; j++)
        move    $5, $0          # j = 0
        ble     $4, 0, $33      # length >= j
        move    $2, $16         # address of a[j]
        addu    $6, $16, $17    # address of a[j+distance]
$32:
 #   9          a[j] = a [j+distance];
        lb      $3, 0($6)       # a[j+distance]
        sb      $3, 0($2)       # a[j]
        addu    $5, $5, 1       # j++
        addu    $2, $2, 1       # address of next a[j]
        addu    $6, $6, 1       # address of next a[j+distance]
        blt     $5, $4, $32     # j < length
$33:
```

The optimized version contains fewer total instructions and fewer instructions that reference memory. Wherever possible, the optimizer replaces load and store instructions (which reference memory) with the faster computational instructions that perform operations only in registers.

4.5.3 Improving C Program Optimization

When you write a C program, you can follow certain guidelines that help you write code that is easier to optimize. Some practices are helpful when you use the `uopt` optimizer. Others are helpful when you use only the limited optimizations provided by the code generator and assembler phases of the compiler. This section explains practices that help `uopt` optimize well and practices that help the code generator and assembler optimize well.

The following practices can help increase optimizing opportunities for `uopt`:

- Avoid using indirect calls.

  Indirect calls (calls that use routines or pointers to functions as arguments) cause unknown side effects (that is, change global variables) that can reduce the amount of optimization. Avoid using indirect calls.

- Use function return values.

  Use function return values instead of reference parameters.

- Use the `do while` statement when possible.

  Use `do while` instead of `while` or `for` when possible. When you use `do while`, the optimizer does not have to duplicate the loop condition to move code from within the loop to outside the loop.

- Avoid using unions for integer and floating point types.

  Avoid unions that cause overlap between integer and floating point data types. Using this type of union keeps the optimizer from assigning the fields to registers.

- Use local variables.

  Avoid using global variables. Minimizing the use of global variables increases optimization opportunities for the compiler. Declare any variable outside of a function as static, unless that variable is referenced by another source file.

- Use value parameters.

  Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers.

- Avoid using aliases.

  Avoid using aliases by introducing local variables to store dereferenced values. (A dereferenced value is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, but local variables are not. Therefore, local variables can be kept in registers. The following three examples show how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code:

  Source code:
  ```
  int len = 10;
  char a[10];
  void
  zero()
    {
    char *p;
    for (p =a; p != a +len; ) *p++ = 0;
    }
  ```
  Generated assembly code:
  ```
  # 8   for (p = a; p != a + len; ) *p++ = 0;
          move    $2, $4          # p = a
          lw      $3, len
          addu    $24, $4, $3
          beq     $24, $4, $33    # a + len != a
  $32:
          sb      $0, 0($2)       # *p = 0
          addu    $2, $2, 1       # p++
          lw      $25, len
          addu    $8, $4, $25
          bne     $8, $2, $32     # len + a != p
  $33:
  ```

To increase the efficiency of the preceding example, you can use one of two methods:

- Use subscripts instead of pointers.

  In the example that follows, the use of subscripting in the procedure `azero` eliminates aliasing. The compiler keeps the value of `len` in a register, which saves two instructions. A pointer is used by the compiler to access the variable a efficiently, even though a pointer is not specified in the source code.

  Source code:
  ```
  void
  azero()
    {
    int i;
    for (i = 0; i != len; i++) a[i] = 0;
    }
  ```
  Generated assembly code:
  ```
  #  14     for (i = 0; i != len; i++) a[i] = 0;
          move    $2, $0          # i = 0
          beq     $3, 0, $35      # len != 0
          la      $14, a
          move    $2, $14
          addu    $4, $3, $14     # a[len]
  $34:
          sb      $0, 0($2)       # *a = 0
          addu    $2, $2, 1       # a++
          bne     $2, $4, $34     # a != a[len]
  $35:
  ```

- Use local variables.

  Specifying `len` as a local variable or formal argument ensures that aliasing can not take place and permits the compiler to place `len` in a register, as shown:

  Source code:
  ```
  char a[10];
  void
  lpzero(len)
    int len;
    {
    char *p;
    for (p = a; p != a + len; ) *p++ = 0;
    }
  ```
  Generated assembly code:
  ```
  #   8      for (p = a; p != a + len; ) *p++ = 0;
          move   $2, $6             # p = a
          addu   $5, $6, $4
          beq    $5, $6, $33        # a + len != a
  $32:
          sb     $0, 0($2)          # *p = 0
          addu   $2, $2, 1          # p++
          bne    $5, $2, $32        # a + len != p
  $33:
  ```

As the previous examples show, using local variables is a slightly more efficient way to eliminate aliasing than using subscripts instead of pointers.

- Write straightforward code.

When you write code, make it straightforward and easy to understand. For example, do not use autoincrement (++) and autodecrement (--) operators within an expression. When you use these operators for their values, rather than for their side effects, you often get bad code. For example:

```
Bad:
while (n--) {
    .
    .
    .
}
Good:
while (n != 0) {
  n--;
    .
    .
    .
}
```

- Use register declarations.

  The compiler automatically assigns variables to registers. However, specifically declaring a register type lets the compiler make more aggressive assumptions when assigning register variables. Therefore, when possible declare variables as `register`.

- Avoid passing addresses.

  Passing addresses can create aliases, force the optimizer to store variables from registers in their home storage locations, and significantly reduce optimization opportunities that would otherwise be performed by the compiler. Do not pass addresses.

- Avoid using a variable number of arguments.

  Avoid functions that take a variable number of arguments. Using a variable number of arguments causes the optimizer to unnecessarily save all parameter registers on entry.

The following practices can help increase optimizing opportunities for the code generation and assembler phases of the compiler:

- Use tables rather than `if-then-else` or `switch` statements. The following shows an example that demonstrates this practice:

  ```
  Good:
  if ( i == 1 ) c = '1';
   else c = '0';
  More efficient:
  c = "01"[i];
  ```

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers, where they remain during execution of the called routine. Therefore, always declare as the first four parameters those variables that are most frequently manipulated in the called routine, with floating-point parameters preceding nonfloating-point parameters.

- Use word-size variables instead of smaller ones if enough space is available. This practice may take more space, but it is more efficient.

- Rely on `libc` functions (for example, `strcpy`, `strlen`, `strcmp`, `bcopy`, `bzero`, `memset`, and `memcpy`). These functions are coded for efficiency.

- Use the unsigned data type for variables wherever possible. Because it knows the unsigned variable will always be greater than or equal to zero (>=0), the compiler can perform optimizations that would not otherwise be possible. Also, the compiler generates fewer instructions for multiply and divide operations that use a power of 2.

For example:

```
int i;
unsigned j;
     .
     .
     .
return i/2 + j/2;
```

The compiler generates four instructions for the signed i/2 operations:

```
000000 bgez     r14, 0xC
000004 move     r1, r14
000008 addiu    r1, r1, 1
00000c sra      r15, r1, 1
```

By contrast, the compiler generates only one instruction for the unsigned j/2 operation:

```
000010 srl      r24,r5,1    # j / 2
```

In the preceding examples, the i/2 expression is less efficient than the j/2 expression.

4.5.4 Improving Pascal Program Optimization

When you write a Pascal program, you can follow certain guidelines that help you write code that is easier to optimize. Some practices are helpful when you use the `uopt` optimizer. Others are helpful when you use only the limited optimizations provided by the code generator and assembler phases of the compiler. This section explains practices that help `uopt` optimize well and practices that help the code generator and assembler optimize well.

The following practices can help increase optimizing opportunities for `uopt`:

- Avoid indirect calls.

  Indirect calls (calls that use routines or pointers to functions as arguments) cause unknown side effects (that is, change global variables) that can reduce the amount of optimization. Therefore, avoid using indirect calls.

- Use function return values.

  Use function return values instead of reference parameters.

- Use the `repeat` statement when possible.

  Use `repeat` instead of `while` or `for` when possible. When you use `repeat`, the optimizer does not have to duplicate the loop condition to move code from within the loop to outside the loop.

- Avoid using certain variant records.

  Avoid variant records that cause overlap between integer and floating point data types. Avoiding this type of variant record keeps the optimizer from assigning the fields to registers.

- Use local variables.

  Avoid using global variables. Minimizing the use of global variables increases optimization opportunities for the compiler.

- Use value parameters.

  Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers.

- Use packed arrays only when space is limited.

  Packed arrays prevent the moving of induction expressions from within a loop to outside the loop. Use them only when you need to save space.

The following practices can help increase optimizing opportunities for the code generator and assembler phases of the compiler:

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers, where they remain during execution of the called routine. Therefore, always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating-point parameters preceding nonfloating-point parameters.
- Use word-size variables instead of smaller ones if enough space is available. This may take more space, but it is more efficient.
- Use predefined functions as much as possible. For example, use `max` and `min` rather than `if-then-else` statements and use `shift` and `bitwise` instead of `div` and `mod`.

4.5.5 Optimizing Your Program Fully

When you optimize your program fully, the `uld` and `umerge` phases of the compiler merge the separate modules in your program into a single module. The `uopt` optimizer is then able to optimize across the modules in your program.

To fully optimize your program, invoke the `uopt` optimizer using the `-O3` option on the `cc` command line. This section provides examples of compiling and optimizing a multimodule program. One example demonstrates compiling and optimizing the modules simultaneously, while the other example demonstrates compiling modules separately and optimizing them later. The examples provided in this section assume that the program `myprogram` consists of three files: `a.c`, `b.c`, and `c.c`.

To compile and fully optimize all three files, enter the following command:

```
% cc -O3 -o myprogram a.c b.c c.c
```

This example causes the compiler to compile, load, merge, and optimize the three modules as a single unit.

You may want to compile the modules of your program separately and then optimize them using the `-O3` option. Follow these steps to optimize modules you compile separately:

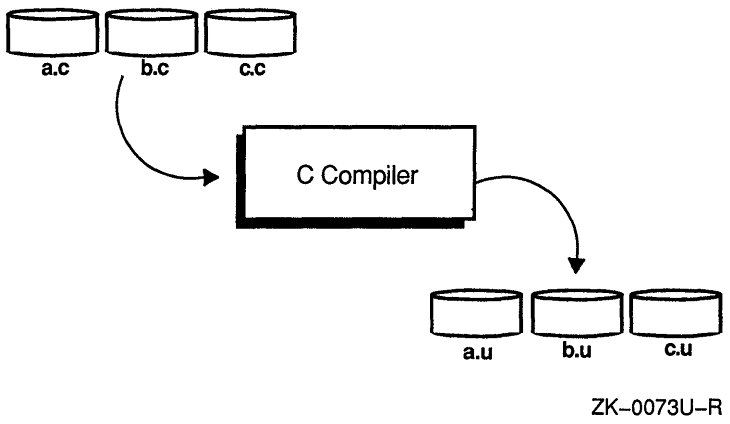
1. Use the `-j` option when you compile each source file, as shown in the following

example:

```
% cc -j a.c
% cc -j b.c
% cc -j c.c
```

The `-j` option causes the compiler driver to produce a `.u` file (the standard compiler front-end output, which is made up of `ucode`, an internal language used by the compiler). None of the remaining compiler phases are executed, as is illustrated by Figure 4-2.

Figure 4-2: Output From the -j Compiler Option

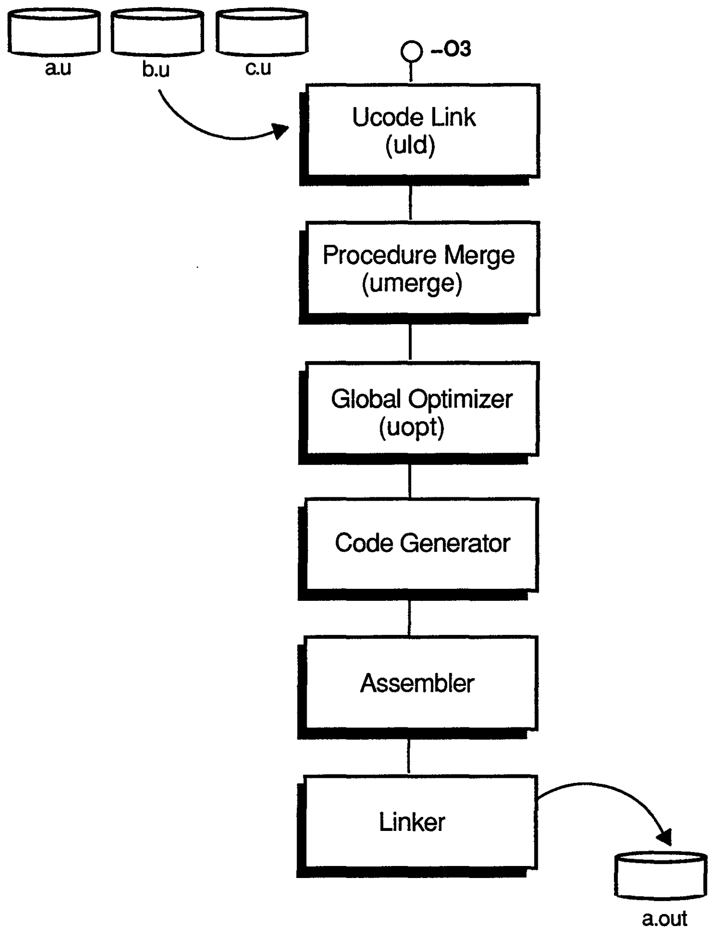


2. Perform optimization and complete the compilation process, by entering the following command:

   ```
   % cc -O3 -o myprogram a.u b.u c.u
   ```

   Figure 4-3 illustrates the results of executing this command.

Figure 4-3: ucode File Optimization



ZK-0072U-R

4.5.6 Optimizing Large Programs

Because compilation time increases by the square of the program module size, the compiler enforces an upper limit on the size of a module that can be optimized. By default, the limit is 500 basic blocks.

To ensure that all modules are optimized regardless of their size, specify the `-O` *`limit`* option when you compile your program. Replace *`limit`* with the maximum size, in basic blocks, of any module that you want the compiler to optimize. If a routine is larger in basic blocks than the default or current `-O` *`limit`* value, the `uopt` optimizer warns you that the routine is too large. In addition, `uopt` displays a message that contains the minimum `-O` *`limit`* value to specify for the routine to be optimized.

The following example demonstrates how to specify the maximum size of modules to be optimized:

```
% cc -O[750] a.c b.c c.c
```

This `cc` command specifies optimizing modules that contain 750 or less basic blocks.

4.6 Optimizing Programs on a VAX System

When you optimize your code, your program runs faster and its object is smaller in size. Using the the optimizer can also speed up development time. For example, your coding time can be reduced if you let the optimizer relate programming details to execution time efficiency. This time savings lets you focus on the more crucial global structure of your program. Moreover, programs often yield code sequences that can be optimized regardless of how well you write your source program.

On ULTRIX systems running on the VAX architecture, the optimizer is `c2`. The `c2` optimizer is invoked when you use the `-O`, option compiler command line. (Note that the VAX FORTRAN/ULTRIX and VAX C/ULTRIX products do not use the `c2` optimizer. See your documentation for those products for information on optimizing VAX FORTRAN/ULTRIX and VAX C/ULTRIX programs.) If you omit the `c2` option from the compiler command line, limited optimizations are performed. The code generator phase of the compiler performs the limited optimizations.

The following sections provide an overview of issues to consider before you optimize a program and ways in which you can write your C program so that `c2` and the code generator optimize well.

4.6.1 Things to Consider Before You Optimize a Program

Before you optimize your program, be sure it is fully developed and is relatively error free. Although the optimizer does not alter the flow of control within a program, it may move operations so that the object code does not correspond to the source code. These changed sequences of code may create confusion when you use the debugger.

If you are writing Pascal programs, be aware that the `-C` option of the Pascal compiler inhibits some optimizations. The `-C` option performs bounds checking. Unless bounds checking is crucial, do not specify the `-C` option when you optimize a Pascal program.

4.6.2 Improving C Program Optimization

The following recommendations can help increase optimizing opportunities for the optimizer (`c2`).

- Avoid using indirect calls.

  Indirect calls (calls that use routines or pointers to functions as arguments) cause unknown side effects (that is, change global variables) that can reduce the amount of optimization. Avoid using indirect calls.

- Use function return values.

  Use function return values instead of reference parameters.

- Avoid using certain unions.

Avoid unions that cause overlap between integer and floating point data types. Using this type of union keeps the optimizer from assigning the fields to registers.

- Use local variables.

  Avoid using global variables. Minimizing the use of global variables increases optimization opportunities for the compiler. Declare any variable outside of a function as static, unless that variable is referenced by another source file.

- Use value parameters.

  Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers.

- Avoid using aliases.

  Avoid using aliases by introducing local variables to store dereferenced values. (A dereferenced value is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables are not. Therefore, local variables can be kept in registers. The following three examples show how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code:

Source code:
```
int len = 10;
char a[10];
void
zero()
  {
  register char *p;
  for (p =a; p != a +len; ) *p++ = 0;
  }
```
Generated assembly code:
```
# for (p = a; p != a+len; ) *p++ = 0;
        moval    _a,r11           # p = a
        jbr      L20
L2000001:
        clrb     (r11)+           # *p++ = 0
L20:    addl3    _len,$_a,r0      # a+len
        cmpl     r11,r0           # p != a+len
        jneq     L2000001
```

To increase the efficiency of the preceding example, you can use one of two methods:

- Use subscripts instead of pointers.

  In the following example, the use of subscripting in the procedure `azero` eliminates aliasing. The compiler keeps the value of `len` in a register, which saves two instructions. A pointer is used by the compiler to access `a` efficiently, even though a pointer is not specified in the source code.

  Source code:
  ```
  void
  azero()
    {
    register int i;
    for (i = 0; i != len; i++) a[i] = 0;
    }
  ```
  Generated assembly code:
  ```
  # for (i = 0; i != len; i++) a[i] = 0;
          clrl     r11              # i = 0
          jbr      L20
  L2000001:
  ```

```
        clrb    _a[r11]         # a[i] = 0
        incl    r11             # i++
L20:    cmpl    r11,_len        # i != len
        jneq    L200000
```

- Use local variables.

  In the following example, specifying `len` as a local variable or formal argument ensures that aliasing cannot take place and permits the compiler to place `len` in a register.

  Source code:

  ```
  char a[10];
  void
  lpzero(len)
    register int len;
    {
    register char *p;
    for (p = a; p != a + len; ) *p++ = 0;
    }
  ```

  Generated assembly code:

  ```
  # for (p = a; p != a+len; ) *p++ = 0;
          movl    4(ap),r11       # p = a
          moval   _a,r10          # register p
          jbr     L19
  L2000001:
          clrb    (r10)+          # *p++ = 0
  L19:    addl3   r11,$_a,r0      # a+len
          cmpl    r10,r0          # p != a+len
          jneq    L2000001
  ```

As the previous examples show, using local variables is a slightly more efficient way to eliminate aliasing than using subscripts instead of pointers.

- Write straightforward code.

  When you write code, make it straightforward and easy to understand. For example, do not use autoincrement (++) and autodecrement (--) operators within an expression. When you use these operators for their values, rather than for their side effects, you often get inefficient code. For example:

  Bad:

  ```
  while (n--) {
      .
      .
      .
  }
  ```

  Good:

  ```
  while (n != 0) {
    n--;
      .
      .
      .
  }
  ```

- Use register declarations.

  Because the compiler will not place a variable in a register unless directed to do so, declare variables as `register` whenever possible.

The following practices can help increase the optimizing opportunities for the code generator:

- Use tables rather than `if-then-else` or `switch` statements. The following example shows how the use of tables makes code more efficient:

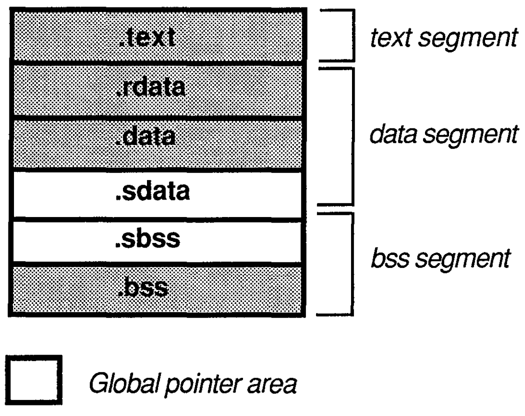
```
Good:
if ( i == 1 ) c = '1';
 else c = '0';
More efficient:
c = "01"[i];
```

- Rely on `libc` functions (for example, `strcpy`, `strlen`, `strcmp`, `bcopy`, `bzero`, `memset`, and `memcpy`). These functions are coded for efficiency.

4.7 Controlling the Size of Global Pointer Data on RISC Systems

Global pointer data is constants and variables that the compiler places in the `.sdata` and `.sbss` portions of the `data` and `bss` segments shown in Figure 4-4. This area is referred to as the global pointer area.

Figure 4-4: Global Pointer Area



ZK-0076U-R

(The `.rdata`, `.data`, and `.sdata` sections contain initialized data, and the `.sbss` and `.bss` sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros.)

In general, the compiler creates two machine instructions to access a global value. However, by using a register as a global pointer (called `$gp`), the compiler creates the 65,536-byte global pointer area where a program can access any value with a single machine instruction – half the number of instructions required without a global pointer.

To maximize the number of individual variables and constants that a program can access in the global pointer area, the compiler first places those variables and constants that take the fewest bytes of memory. By default, the variables and constants occupying eight or fewer bytes are placed in the global pointer area, and those occupying more than eight bytes are placed in the `.data` and `.bss` sections.

4.7.1 Limiting the Size of Global Pointer Data

The more data that the compiler places in the global pointer area, the faster a program executes. However, if the data to be placed in the global pointer area exceeds 65,536 bytes, the linker displays an error message and does not create an executable object file. In this case, you need to recompile your program and use the `-G` option to reduce the use of global data.

For most programs, the 8-byte default produces optimal results. However, the compiler provides the `-G` option to let you change the default size. For example:

```
% cc -G 12 a.c b.c c.c
```

This command causes the compiler to place only those variables and constants that occupy 12 or fewer bytes in the global pointer area.

The compiler places some variables in the global pointer area regardless of the setting specified by the `-G` option. For example, a program written in assembly language might contain `.sdata` directives that cause variables and constants to be placed into the global pointer area regardless of size. Moreover, the `-G` option does not affect variables and constants in libraries and objects compiled beforehand.

To alter the allocation size for the global pointer area for data from these objects, you must recompile them and specify the `-G` option and the desired value.

4.7.2 Obtaining Optimal Global Data Size

Two potential problems exist in specifying a maximum size in the `-G` option:

- Using a value that is too small can reduce the speed of the program.
- Using a value that is too large can cause more than the maximum of 65,536 bytes to be placed in the data area, which creates an error condition and produces an unexecutable object module.

The `-bestGnum` linker option helps you avoid these problems by predicting an optimal value to specify for the `-G` option. This section provides examples of using the `-bestGnum` option and the related `-nocount` and `-count` options.

In the following example, the compiler displays a message that provides the best value for `-G`:

```
% pc -bestGnum myprogram.p
All data will fit into the global pointer area
Best -G num value to compile with is 80 (or greater)
```

Because all data fits into the global pointer area, no recompilation is necessary.

Consider the following example, which specifies 70,000 as the maximum size of a data item to be placed in the global pointer area:

```
% pc ersatz.p -G 70000 -bestGnum
gp relocation out-of-range errors have occurred and bad object file
produced (corrective action  must be taken)
Best -G num value to compile with is 1024
```

In this example, the linker does not produce an executable load module and recommends a recompilation as follows:

```
% pc real.p -G 1024
```

When you use the `-bestGnum` option without using `-nocount` or `-count`, the compiler assumes that you cannot recompile any libraries to which it would link automatically. Because you cannot recompile these libraries, you cannot specify a new value for them using the `-G` option. The linker ignores libraries you cannot recompile when predicting the optimal value for the `-G` option.

If you link to system-supplied libraries other than those that are included automatically, you must specify `-nocount` before the library, as shown in the following example:

```
% cc -bestGnum myprogram.c -nocount -lm
```

Because the system does not automatically link with the `lm` library and because you cannot recompile the library, the linker should not count that library when predicting the best value for `-G`. The `-nocount` option ensures that the linker ignores the `lm` library.

You can explicitly specify that the linker both include and exclude specific libraries in predicting the `-G` value, as shown in the following example:

```
% cc -o plotter -bestGnum plotter.o -nocount libieee.a-count \
liblaser.a
```

In this example, the linker assumes that you cannot recompile the `libieee.a` library and that it continues to occupy the same space in the global pointer area. The compiler assumes that you can recompile `plotter.o` and `liblaser.a`, and it produces a recommended `-G` value to use on recompilation.

4.7.3 Allocating the Global Pointer Area

If your program contains several modules and the data for all modules is too large to fit in the global pointer area, you can allocate the global pointer area to the module that is most active. For example, suppose your program consists of modules `a.c`, `b.c`, and `c.c`. You discover by using `prof` that most of the execution time is spent in module `a.c`. To make your program efficient, allocate the global pointer area to the `a.c` module, as shown:

```
% cc -c -G 1000 a.c
% cc -c -G 0 b.c c.c
```

These commands cause the compiler to allow only the data from the `a.c` module to occupy the global pointer area.

Library Routines and System Calls 5

ULTRIX provides a number of routines you can call from your program. These routines perform programming tasks, such as reading or writing a file, and they perform tasks that control the system, such as mounting a file system. The routines ULTRIX provides are grouped into two major types: library routines and system calls.

The library routines are C functions grouped in various archive libraries. System calls are the system primitive routines that are entry points to the ULTRIX kernel. Like library routines, system calls are callable from C.

When possible, use a library routine or system call instead of writing a routine of your own to perform a particular task. Using system calls and library routines saves you coding and debugging time. Because most library routines and system calls are standard across UNIX systems, using them makes your program more portable.

This chapter provides information about ULTRIX library routines and system calls, compilation and linking considerations, and the contents of the various libraries. This chapter also describes using ULTRIX input and output routines to manipulate data.

For information about ULTRIX library routines beyond what is given in this chapter, see Section 3 in the *ULTRIX Reference Pages.* For more information about the system calls, see Section 2 in the *ULTRIX Reference Pages.*

5.1 Compiling and Linking Considerations

When you compile and link your program, the compilers for C, Pascal, FORTRAN, and perhaps other languages automatically attempt to resolve references by searching the standard C library, `libc.a`. This library contains all the system calls, the general purpose library routines, and the following groups of special library routines:

- Routines that perform standard input and output
- Routines that control the internet network
- Routines that control the X/Open Transport Interface
- Routines that control the Yellow Pages Service (YP)

In addition to `libc.a`, the system contains other libraries with which you can link your program. For example, if your program uses the interface to Kerberos, you must link it with the `libkrb.a`, `libknet.a`, and `libdes.a` libraries. For information on linking with a library other than `libc`, see Section 2.4.3.

When you call a library routine or system call in your program, you must declare the routine or system call, just as you would any other routine in your program. ULTRIX provides header files that contain declarations of the library routines and system calls. For example, if you call a standard I/O routine that is in `libc`, you must include the header file `<stdio.h>` in your program. You might also need to include other header files, depending on which routine you call. See the reference page for a specific library routine or system call to determine what header files other than `<stdio.h>` you need to include. For information on including header files in your program, see Section 2.2.

5.2 The C Library

The standard C library, `libc.a`, contains the system calls, many commonly used routines, and groups of special library routines.

The following list refers you to other sources for information about routines in `libc.a`:

- See Section 5.4 for information on performing input and output with `libc` routines.
- See *Guide to the X/Open Transport Interface* for information on controlling the X/Open Transport Interface using `libc` routines.
- See Chapter 6 for information on controlling interprocess communication using `libc` routines.

5.2.1 Character Processing Routines and Macros

Table 5-1 describes C library routines associated with character processing. These routines require the inclusion of header file `<ctype.h>`.

Table 5-1: Character Processing Routines and Macros

Name	Description
`isalnum()`	Tests for an alphanumeric ASCII character.
`isalpha()`	Tests for an alphabetic character.
`isascii()`	Tests for an ASCII character.
`iscntrl()`	Tests for a control character.
`isdigit()`	Tests for a digit.
`isgraph()`	Tests for a graphic ASCII character (any printing character other than a space).
`islower()`	Tests for a lowercase letter.
`isprint()`	Tests for a printing ASCII character (including a space).
`ispunct()`	Tests for a punctuation character (printing character that is nonalphanumeric and greater than octal 40).
`isspace()`	Tests for one of the following white space characters: space, form feed, new line, carriage return, horizontal tab, or vertical tab.
`isupper()`	Tests for an uppercase letter.
`isxdigit()`	Tests for a hexadecimal digit.
`toascii()`	Converts an integer to an ASCII character.
`tolower()` `_tolower()`	Converts an uppercase letter to lowercase.
`toupper()` `_toupper()`	Converts a lowercase letter to uppercase.

Related to the `toascii()` conversion routine, the math library routines `atoi()`, `atol()`, `atof()`, `strtol()`, `strtoul()`, and `strtod()` convert string data to various numeric forms.

5.2.2 Standard I/O Routines

Table 5-2 describes C library routines associated with standard I/O operations on files and with file access. These routines require the inclusion of header file `<stdio.h>`. Standard I/O routines are also described in `intro`(3s).

Table 5-2: Standard I/O Routines Related to Files and File Access

Name	Description
`fclose()`	Flushes buffers of the specified stream and closes the specified stream, including disassociating the stream from the file.
`fflush()`	Flushes buffered data of the specified stream.
`fdopen()`	Associates a stream with an open file.
`fopen()`	Opens the specified file, including associating a stream with the file. You need to specify the file mode to indicate the types of operations to be performed.
`freopen()`	Opens the specified file after attempting to close the file associated with the stream.

`setbuf()`
`setvbuf()`
	Associates a buffer with an input or output file.

`tmpfile()`	Creates a temporary file.

`tmpnam()`
`tempnam()`
	Generates a valid file name for a temporary file.

Related routines include the C library routine `remove()` and the system calls `creat()`, `rename()`, and `unlink()`.

Table 5-3 describes standard I/O routines associated with formatted I/O and character I/O. These routines require the inclusion of header file `<stdio.h>`.

Table 5-3: Standard I/O Routines for Formatted I/O and Character I/O

Name	Description
Formatted I/O Routines:	
`fprintf()`	Writes output to the specified stream using the specified format.
`fscanf()`	Reads input from the specified stream using the specified format.
`printf()`	Writes output to `stdout`.
`scanf()`	Reads input from `stdin`.

Table 5-3: (continued)

Name	Description
`sprintf()`	Writes output to the specified string, terminated by a null character.
`sscanf()`	Reads input from a specified string.
`vfprintf()`	Similar to `fprintf()`, except a variable argument list is used (requires an additional header file).
`vprintf()`	Similar to `printf()`, except a variable argument list is used (requires an additional header file).
`vsprintf()`	Similar to `sprintf()`, except a variable argument list is used (requires an additional header file).
Character I/O Routines:	
`fgetc()`	Reads the next character from the specified input stream (as an integer). This is the function used by the macro `getc()`.
`fgets()`	Reads up to the specified number of characters from the specified stream.
`fputc()`	Writes the next character to the specified output stream.
`fputs()`	Writes the specified string to the specified output stream.
`getc()`	Similar to `fgetc()`, except it is implemented as a macro.
`getchar()`	Similar to `fgetc()`, but uses `stdin`.
`gets()`	Reads characters from `stdin` into an array.
`putc()`	Similar to `fputc()`, except it is implemented as a macro.
`putchar()`	Similar to `fputc()`, except it is directed only to `stdout`.
`puts()`	Writes a string to the `stdout` stream.
`ungetc()`	Pushes the specified character back into the input stream and leaves the stream at the position before the inserted character.

Table 5-4 describes the standard I/O routines associated with direct I/O, file positioning, and error handling. Like other standard I/O routines, these routines require the inclusion of header file <`stdio.h`>.

Table 5-4: Standard I/O Routines for Direct I/O, File Positioning, and Error Handling

Name	Description
Direct I/O Routines:	
`fread()`	Reads the specified number of data elements into an array from the specified stream.
`fwrite()`	Writes the specified number of data elements from an array onto the specified stream.
File Positioning Routines:	
`fgetpos()`	Stores and returns the current value of the file position indicator for the specified stream.

Table 5-4: (continued)

Name	Description
`fseek()`	Sets the file position indicator to the specified offset for the specified stream.
`fsetpos()`	Sets the file position indicator for the specified stream. Usually used with `fgetpos()`.
`ftell()`	Returns the current file position indicator for the specified stream.
`rewind()`	Sets the current file position indicator for the specified stream to the beginning of the file.
Error Handling Routines:	
`clearerr()`	Clears the end-of-file and error indicators for the specified stream.
`feof()`	Tests whether the end-of-file indicator is set for the specified stream.
`ferror()`	Tests the error indicator for the specified stream. If an error is present, use `errno` or the `strerror()` routine to return the value.
`perror()`	Maps the error number to a error message and prints it to `stderr`.

Routines that might be used with error handling routines include the C library functions `setjmp()` and `longjmp()`, which allow non-local transfer of control.

5.2.3 Memory Management, Environment, and General Functions

Table 5-6 describes C library routines associated with pseudo-number generation, memory management, environment and process control, sorting and searching, and integer arithmetic. Most of these routines require the inclusion of header file `<stdlib.h>`.

Table 5-5: General Routines

Name	Description
Pseudo-Random Number Generation Routines:	
`rand()`	Returns a sequence of pseudo-random integers based on the specified seed. Should be preceded by a call to `srand()`.
`srand()`	Sets the specified argument as a seed. This seed determines the values returned by `rand()`.
Memory Management Routines:	
`calloc()`	Allocates a zero-filled area of memory using a specified number of units of the same size.
`free()`	Deallocates an area of memory previously allocated by `calloc()`, `malloc()`, or `realloc()`.
`malloc()`	Allocates a contiguous space in memory of the specified size.
`realloc()`	Changes the size of allocated memory to the specified size.

Table 5-5: (continued)

Name	Description
Environment and Process Control Routines:	
`abort()`	Causes abnormal program termination by use of the SIGABRT (SIGIOT) signal (which may be caught).
`exit()`	Causes normal program termination to occur.
`getenv()`	Searches an environment variable list for a specified string.
`popen()`	Initiates pipe I/O and executes a Bourne shell command. Terminate using `pclose()`.
`putenv()` `setenv()`	Sets an environment variable.
`system()`	Passes the specified string as a command for shell execution.
`unsetenv()`	Unsets an environment variable.
Searching and Sorting Routines:	
`bsearch()`	Used for searching and sorting array elements in conjunction with a user function. The `bsearch()` routine returns a pointer to the matching element of the array. The user function examines two arguments and returns a value indicating whether the first is greater than the second, the second is greater than the first, or that they are equal.
`hsearch()`	Performs a hashed table search. Used with `hcreate()` and `hdelete()`. Requires `<search.h>`.
`tsearch()`	Initiates a binary tree search. Used with `tfind()` and related routines. Requires `<search.h>`.
`qsort()`	Sorts an array of the specified number of elements.
Integer Arithmetic Routines:	
`abs()`	Returns an absolute value for the specified integer.
`div()`	Performs division of two integer values, returning the quotient and remainder.
`labs()`	Returns the absolute value of the specified long integer.
`ldiv()`	Performs division of two long integer values, returning the quotient and remainder as long integers.

5.2.4 String Operations

Table 5-6 describes C library routines associated with string operations. These routines require the inclusion of header file `<string.h>`.

Table 5-6: String Processing Routines

Name	Description
`memcpy()` `memmove()`	Copies the specified number of characters from one area of memory to another. Using `memmove()` uses an intermediate buffer if the two areas overlap.

Table 5-6: (continued)

Name	Description
`memchr()`	Returns a pointer to the first occurrence of the specified character, examining the specified number of characters from an area of memory.
`memcmp()`	Compares two arguments lexicographically in memory, looking at the specified number of characters, and indicates whether one is greater than, less than, or equal to the other.
`memset()`	Sets the first specified number of characters in memory to the value of the specified character.
`strcpy()`	Copies one specified string to another string.
`strncpy()`	Copies one specified string to another string, up to the specified number of characters.
`strcat()`	Copies (appends) a specified string to the end of another string.
`strncat()`	Copies (appends) a specified string to the end of another string, up to the specified number of characters.
`strcmp()`	Compares two arguments and indicates whether one is greater than, less than, or equal to the other.
`strcoll()`	Compares two strings lexicographically, using collating information defined in the program's locale.
`strncmp()`	Compares two arguments up to the specified number of characters and indicates whether one is greater than, less than, or equal to the other.
`strxfrm()`	Transforms one string into another string.
`strchr()`	Locates the first occurrence of the specified character in a string.
`strcspn()`	Returns the length of a segment of a string that does not contain any of the characters in a second string.
`strpbrk()`	Locates the position of a string that contains any of the characters of the second string.
`strrchr()`	Locates the last occurrence of a specified character in a string.
`strspn()`	Returns the length of a string that consists of only characters from the second string.
`strstr()`	Locates the first occurrence of a specified series of characters in a string.
`strtok()`	Breaks a specified string into a sequence of tokens.
`strerror()`	Returns a pointer to the message text for a given error number.
`strlen()`	Returns the length of a string.

Certain string handling routines are also provided in the Internationalization library, including routines related to multiple byte strings, which include `mblen()`, `mbtowc()`, `mbstowcs()`, `setlocale()`, `wctomb()`, and `wcstombs()`.

5.2.5 Date and Time Routines

Table 5-7 describes the C library routines associated with date and time conversion, and time processing. These routines require the inclusion of header file `<time.h>`.

Table 5-7: Date and Time Processing Routines

Name	Description
Time Conversion Routines:	
`asctime()`	Converts a time structure to ASCII format.
`ctime()`	Converts integer time to ASCII format.
`gmtime()`	Returns pointer to a time structure, using GMT.
`localtime()`	Returns pointer to a time structure, using local time.
`strftime()`	Converts time to ASCII format using a conversion specifier.
`tzset()`	Sets the local time zone.
Time Processing Routines:	
`clock()`	Returns the processor time used.
`difftime()`	Returns the difference between two times.
`mktime()`	Converts a time structure to calendar time.
`time()`	Returns the current calendar time.

5.2.6 System Calls and Other C Library Routines

The system calls allow you to access entry points to the kernel from your program to perform system tasks. For example, the system calls allow you to control sockets, control processes (such as `fork()`), handle signals (such as `sigvec()`), return file status (such as `fstat()`), perform basic I/O (as described in Section 5.4), turn accounting on and off (such as `acct()`), mount and unmount file systems (such as `mount()`), get or set the system clock, and so on.

Other C library routines not listed previously in this section perform a variety of tasks, including routines to execute a file (such as `execl()`), perform floating-point conversion (such as `ftoi()`), change (usually reduce) process priority (such as `nice()`), suspend program execution for a specified interval (such as `sleep()`), and so on. Some of these library routines are typically used in combination with system calls.

5.3 Other Commonly Used Library Routines

The following sections describe many of the library routines not described in Section 5.2.

5.3.1 The Standard Conformant Function Library

ULTRIX provides a function library, `libcP.a`, that conforms to the POSIX and X/Open standards. To use this library, you must link with it, in addition to linking with `libc.a`.

It is important to be aware of differences between the standard-conformant functions in `libcP.a` and the functions in `libc.a`. Table 5-8 lists the functions that differ and explains how the `libcP` functions differ from `libc` functions.

Table 5-8: Standard Conformant Library Functions That Differ from C Library Functions

libcP Function	Differences from libc Function
`abort()`	Closes open files before aborting the process with a SIGABRT signal.
`ctermid()`	Returns a null string if the program has no controlling terminal.
`cuserid()`	Uses the effective user ID, instead of the login user ID.
`fclose()`	Seeks to the byte following the last one your program read or wrote before closing the file.
`fflush()`	Writes buffers even if the file is a read-only file.
`fopen()` `fdopen()` `freopen()`	Causes the ``a'' and ``a+'' mode strings to append with no overwrite.
`nice()`	Returns the new priority value minus NZERO. NZERO is the default process priority as defined in `<limits.h>`. On ULTRIX systems, NZERO is 20.
`opendir()`	Sets the FD_CLOSEXEC flag on the type *`DIR`*.
`printf()` `fprintf()`	On success, returns the number of characters printed.
`scanf()`	Treats the E, G, and X conversion codes the same as the e, g, and x conversion codes.
`sleep()`	Can be interrupted by signals.
`sprintf()`	Returns the number of characters formatted. This return value difference affects the syntax of the function call. (See the *ULTRIX Reference Pages* for more information.)
`tzset()`	Defines the timezone and daylight global variables, which you must declare as *`long`* and *`int`*, respectively.
`ungetc()`	Clears the EOF indicator for the stream.

5.3.2 The Curses Library

The X/Open curses library, `libcursesX.a`, contains routines that perform screen management tasks. The library allows you to perform common terminal-dependent functions without being aware of the detailed description of the current teminal. For information on using the curses library, see *Guide to X/Open curses Screen Handling*.

5.3.3 The Internationalization Library

The internationalization library, `libi.a`, contains routines that provide a convenient method of writing applications so that they operate in the application user's natural languages. You can use the library to display output that is formatted correctly for the application user, even if users in several countries use the same application. For information on using the internationalization library, see the *Guide to Developing International Software.*

5.3.4 The Kerberos Library

The Kerberos libraries are `libkrb.a`, `libknet.a`, `libdes.a`, and `libacl.a`. These libraries authenticate changes made to messages that applications send across a TCP/IP network and protect against the unauthorized modification of such messages. For information on using the Kerberos library, see the *Guide to Kerberos.*

5.3.5 The Mathematical Library

The mathematical library, `libm.a`, provides functions that are useful for performing mathematical equations. For example, the math library provides a function for calculating the inverse hyperbolic function of a real value, performing bessel operations, and so on.

For more information, refer to `intro`(3m).

5.3.6 The Network Computing System Library

The Network Computing System (NCS) Library contains routines that allow you to develop distributed applications. When you develop distributed applications, you usually do not use many of the routines directly. Instead, you write interface definitions in Network Interface Definition Language (NIDL) and use the NIDL Compiler to generate most of the required calls to the library.

For more information about developing distributed applications, see *DECrpc Programming Guide.*

5.3.7 Optional Product Libraries

Other libraries may exist on your system and may be associated with optional products, such as languages or windowing systems. For instance, FORTRAN provides section 3f library routines that simplify calling section 3 routines (written in C) from FORTRAN. Similarly, other optional Digital products may provide separate libraries.

5.4 System I/O and Standard I/O

ULTRIX has two groups of routines for performing I/O. These groups are called system I/O (system calls) and standard I/O (library routines).

System I/O routines:

- Are ULTRIX system calls to the kernel
- Use file descriptors for file access
- Are documented in Section 2 of the *ULTRIX Reference Pages*

Standard I/O routines:

- Are contained in `/usr/lib/libc.a`
- Use a pointer to a `FILE` structure (defined in `<stdio.h>`) for file access

- Call system I/O routines, but are faster than system I/O for small sequential reads (see Figure 5-1)
- Are documented in Section 3 of the *ULTRIX Reference Pages*

Figure 5-1 compares file access using system I/O to file access using standard I/O. The `read()` system call places a specified file in buffer cache, but `fread()` places it in buffer cache and process cache. Reading a file using system I/O's `read()` routine requires a trap to the kernel. Standard I/O's `fread()` routine reads from process cache, requiring no kernel trap, and is therefore faster for small sequential reads.

Figure 5-1: System I/O Versus Standard I/O: File Reading

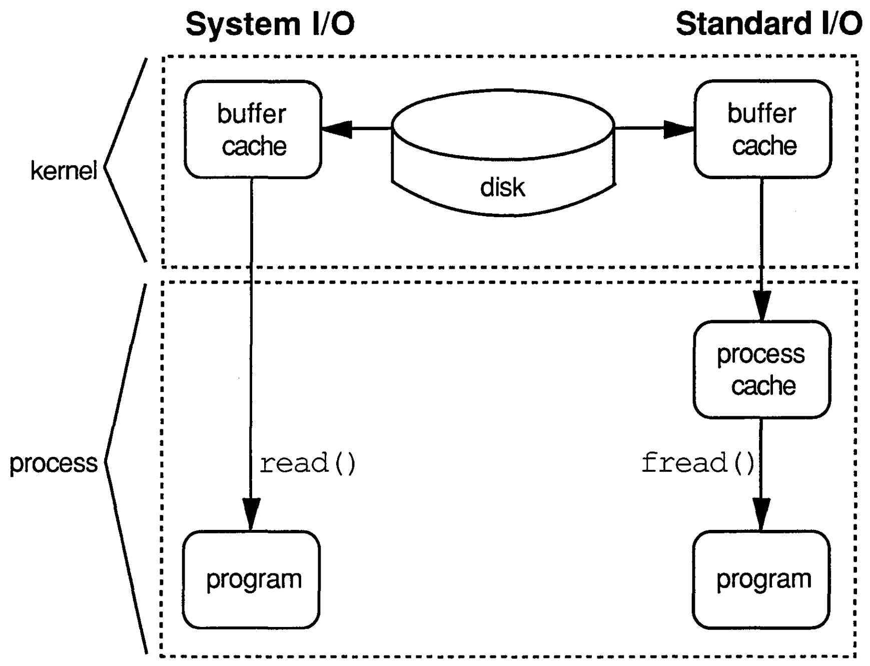


Table 5-9 lists standard I/O and system I/O routines.

Table 5-9: System I/O and Standard I/O

Action	System I/O	Standard I/O
File descriptor declaration	`int fd`	None
`FILE` structure pointer declaration	None	`#include <stdio.h>` `FILE *fp`
Open a file	`open()`	`fopen()`
Close a file	`close()`	`fclose()`
Read from a file	`read()`	`fread(), fgets(), fscanf()`
Write to a file	`write()`	`fwrite(), fputs(), fprintf()`
Position the file pointer within a file	`lseek()`	`fseek()`

5.4.1 File I/O

The program in Example 5-1 performs file I/O using standard I/O routines.

Example 5-1: Using Standard I/O Routines

```
/* standard_io.c  This program uses standard I/O routines to create a
                  data file in the working directory, write 3 structures
                  to the data file, and read back the second structure.
*/
#include <stdio.h>    /* Contains the definition of the FILE structure */

main()
{
    FILE *fp;                           /* Pointer to a FILE structure */
    struct rec
           { char type;             /* Some meaningless character data */
             short buf[10];           /* Some meaningless integer data */
           } rec_1;                 /* A record to be written and read */
    int i, j,                                         /* Loop counters */
        status;                                     /* Status variable */

    fp = fopen("data_file.txt",         /* Create or open data_file.txt */
               "w+");               /* Open it for reading and writing */
    if ( fp == NULL )              /* fopen() returns NULL for failure */
        perror("standard_io.c: fopen"), exit(1);

    for (i = 0 ; i < 3 ; i++)    /* Write 3 structures to data_file.txt */
    {
        for (j = 0 ; j < 10 ; j++)
            rec_1.buf[j] = i + 1;     /* Fill the buffer with integers */
        rec_1.type = 0x31 + i;                      /* ASCII character */

        status = fwrite(&rec_1,                         /* Write rec_1 */
                        sizeof(rec_1),  /* Size of items to be written */
                        1,                              /* Number of items */
                        fp);                          /* File to write to */
        if ( status == 0 )                                  /* Failure */
            perror("standard_io.c: fwrite"), exit(1);
    }
    status = fseek(fp,             /* Place fp's read-write pointer  */
```

Example 5-1: (continued)

```
                 sizeof(rec_1),     /* this many bytes              */
                 0);                /* from the beginning of the file */
    if ( status == -1 )                                  /* Failure */
        perror("standard_io.c: fseek"), exit(1);

    status = fread(&rec_1,                  /* Store at this location */
                 sizeof(rec_1),             /* an item of this size   */
                 1,                                  /* Just one item */
                 fp);                          /* Read from this file */
    if ( status == 0 )                                   /* Failure */
        perror("standard_io.c: fread"), exit(1);

    printf("Second structure:\n\tType:\t%c\n\tContents: ",
           rec_1.type);

    for(i = 0 ; i < 10 ; i++)
        printf(i != 9 ? "%d " : "%d\n", rec_1.buf[i]);

    fclose(fp);                                   /* Close the file */
    exit(0);
}
```

The program in Example 5-2 performs file I/O using system I/O routines. The program in Example 5-2 performs the same tasks as those in Example 5-1, but uses system I/O routines instead of standard I/O routines.

Example 5-2: Using System I/O Routines

```
/* systemio.c  This program uses system I/O routines to create a data
               file in the working directory, write 3 structures to the
               data file, and read back the second structure.
*/
#include <stdio.h>
#include <sys/file.h>

main()
{
    int fd;                                      /* File descriptor */
    struct rec
           { char type;           /* Some meaningless character data */
             short buf[10];         /* Some meaningless integer data */
           } rec_1;               /* A record to be written and read */
    int i, j,                                      /* Loop counters */
        status;                                  /* Status variable */

    fd = open("data_file.txt",                /* Open data_file.txt */
              O_CREAT | O_RDWR,          /* Flags: create, read-write */
              0644);                    /* Create file with this mode */
    if ( fd == -1 )                                       /* Failure */
        perror("systemio.c: open"), exit(1);

    for (i = 0 ; i < 3 ; i++)   /* Write 3 structures to data_file.txt */
    {
        for (j = 0 ; j < 10 ; j++)
            rec_1.buf[j] = i + 1;    /* Fill the buffer with integers */
        rec_1.type = 0x31 + i;                    /* ASCII character */

        status = write(fd,                      /* Write to file fd */
              &rec_1,          /* Copy data starting at this location */
              sizeof(rec_1));                  /* Copy this much data */
        if ( status == -1 )                               /* Failure */
            perror("systemio.c: write"), exit(1);
```

Example 5-2: (continued)

```
    }
    status = lseek(fd,                  /* Place fd's read-write pointer  */
                   sizeof(rec_1),       /* this many bytes                */
                   L_SET);              /* from the beginning of the file */
    if ( status == -1 )                                      /* Failure */
        perror("systemio.c: lseek"), exit(1);

    status = read(fd,                                 /* Read from file fd */
         &rec_1,                                  /* Store at this location */
         sizeof(rec_1));                            /* Read this much data */
    if ( status == -1 )                                      /* Failure */
        perror("systemio.c: read"), exit(1);

    printf("The second structure:\n\tType:\t%c\n\tContents: ",
           rec_1.type, i);

    for(i = 0 ; i < 10 ; i++)
        printf(i != 9 ? "%d " : "%d\n", rec_1.buf[i]);

    close(fd);                                         /* Close the file */
    exit(0);
}
```

5.4.2 Device I/O

Writing to or reading from a device in an ULTRIX system is accomplished using system I/O routines. Instead of data files, device files are used. Every device in an ULTRIX system has a device file associated with it. Reading from or writing to a device file is reading or writing the device; they are the same thing.

All device files for an ULTRIX system are kept in the `/dev` directory. Each device file's name consists of a code that tells what the device is, followed by a number assigned to that device; for example, `/dev/ra13a` is an MSCP disk controller because the file name begins with `ra`. The *Guide to System Environment Setup* lists device codes and their meanings.

There are two different types of device I/O:

- Character mode (also called *raw)*
- Block mode (also called *cooked*).

Devices are usually configured to accept only one mode. Printers and terminals use character I/O, but disks can use either. Generally, an `r` preceding a disk device's file name means the device is character mode. To be certain which mode to use, issue the `file` command, supplying the device file as an argument; for example:

```
% file /dev/nrmt0h
/dev/nrmt0h:    character special (36/12) HSC70 #1 TA78 tape #0 offline
```

In this example, the tape `nrmt0h` is a character mode file (as indicated by the words `character special`).

To write to this device, open it using its file name; for example:

```
fd = open ("/dev/nrmt0h", O_WRONLY);      /* Open the tape for writing */
write (fd, buf, sizeof(buf)); /* Write the contents of buf to the tape */
```

System I/O routines and standard I/O routines perform both character and block device I/O. The type of I/O is determined by the device being accessed. Which routine a program uses depends on the behavior the programmer wants.

5.4.2.1 Character Mode (Raw) Device I/O

In character I/O, device drivers are read and written to directly. Device drivers are called by the reading and writing routines. The buffer cache is not used. If input from a device is going to be processed or manipulated, using `fread()` to read the input into the user's process space might make the most sense. If it is important to the program that output be written to a device before the program continues, using `write()` to make the program wait until the device driver has finished writing might be best.

5.4.2.2 Block Mode (Cooked) Device I/O

In block mode I/O, the reading and writing routines read and write the buffer cache. The kernel calls device drivers as needed to fill or empty the cache.

5.4.3 Controlling Devices with ioctl()

The `ioctl()` system call can be used to control I/O on files, disks, sockets, terminals, and tapes. The `ioctl()` call has the following form:

```
ioctl(fd,       /* An open device file descriptor */
      request,  /* A device-specific request */
      ptr);     /* A pointer to either char or
                   a device-specific structure */
```

The second argument to `ioctl()` is a device-specific request that names the type of action desired, such as TIOCGETC, which gets information about a terminal's current characteristics. All device-specific requests are defined in `<ioctl.h>`.

The third argument to `ioctl()` is a pointer to char or a device-specific structure, which stores information about the device. These structures are written to if the request is to obtain information, or read from if the request is to change device characteristics. The structure used varies with the request made. These structures are defined in various headers, depending on device type, as shown in Table 5-10.

Table 5-10: Headers That Define Structures Used with `ioctl()`

Device	Header That Defines Needed `ioctl()` Structures
Data file	None
Disk	`<sys/dkio.h>`
Generic device	`<sys/devio.h>`
Socket	`<sys/socket.h>`
Tape	`<sys/mtio.h>`
Terminal	`<sys/sgtty.h>`

5.4.3.1 Multibuffered I/O with Character Mode (Raw) Devices

If a program is reading or writing a raw device, the program waits (is blocked) until the I/O is complete. If there are several I/O operations, the waits are serial because there is a single buffer that must be emptied before it can be reused; for example:

```
fd = open ("/dev/nrmt0h", O_WRONLY);
write (fd, buf1, sizeof(buf1)); /* Process waits until
                                   buf1 is written to fd */
write (fd, buf2, sizeof(buf2)); /* Process waits until
                                   buf2 is written to fd */
write (fd, buf3, sizeof(buf3)); /* Process waits until
                                   buf3 is written to fd */
```

Having multiple buffers speeds I/O to raw devices. The `ioctl()` system call with the FIONBUF argument is used for multibuffered I/O; for example :

```
int buffers = 3;                          /* Number of buffers needed */
char data_1[100], data_2[100], data_3[100],
     *ptr = data_3;

ioctl(fd, FIONBUF, &buffers);         /* Create three buffers for I/O */

write(fd, data_1, 100);                                /* No waiting */
write(fd, data_2, 100);                                /* No waiting */
write(fd, data_3, 100);                                /* No waiting */

                  /* Perform other work here while the writes complete */

status = ioctl(fd, FIONBDONE, &ptr);        /* Wait for write to data_3 */

  /* Perform work here that must be done after the writes are complete */
```

Using `ioctl()` with the FIONBDONE argument is not the only way to determine when multiple I/O operations are done; there are three other ways:

```
fcntl(fd, F_SETFL, FASYNC); /* Send SIGIO signal when fd's I/O is done */


fcntl(fd, F_SETFL, FNDELAY);       /* Alter the behavior of ioctl(fd,    */
                                   /* FIONBDONE,...) so that the ioctl() */
                                   /* call does not block, but merely    */
                                   /* returns EWOULDBLOCK                */


select();       /* Wait a maximum specified time for the I/O to finish */
```

Example 5-3 shows a complete program that uses multiple buffers to write to a tape.

Example 5-3: Multibuffered Writing to a Tape

```
/* tape_write.c  Perform multi-buffered output to a raw tape.  Write
                 four buffers of BUFSIZE bytes each to a tape.  The
                 first buffer contains 'A's, the second 'B's,
                 and so forth.
*/
#include <stdio.h>
#include <sys/file.h>
#include <sys/ioctl.h>

#define BUFSIZE 100000

main()
{
    char buf1[BUFSIZE], buf2[BUFSIZE], buf3[BUFSIZE], buf4[BUFSIZE],
         *ptr = buf4;                 /* Pointer to the last buffer */
    long i;                                          /* Loop counter */
    int  fd,                                      /* File descriptor */
         buffers = 4;                   /* Number of buffers needed */

    fd = open("/dev/nrmt0h", O_WRONLY); /* Open tape for write only */
    if ( fd == -1 )
         perror("tape_write.c: open"), exit(1);          /* Failure */

    for (i = 0 ; i < BUFSIZE ; i++)
    {                                  /* Fill the arrays with data */
        buf1[i] = 'A';
        buf2[i] = 'B';
        buf3[i] = 'C';
        buf4[i] = 'D';
    }
    if ( ioctl(fd, FIONBUF, &buffers) == -1 )     /* Create multiple */
         perror("tape_write.c: ioctl FIONBUF"), exit(1); /* buffers */

    if ( write(fd, buf1, sizeof(buf1)) == -1 )     /* Write the four */
         perror("tape_write.c1: write"), exit(1);/* buffers to tape */
    if ( write(fd, buf2, sizeof(buf2)) == -1 )
         perror("tape_write.c2: write"), exit(1);
    if ( write(fd, buf3, sizeof(buf3)) == -1 )
         perror("tape_write.c3: write"), exit(1);
    if ( write(fd, buf4, sizeof(buf4)) == -1 )
         perror("tape_write.c4: write"), exit(1);

    if ( ioctl(fd, FIONBDONE, &ptr) == -1 )  /* Wait for last write */
         perror("tape_write.c: ioctl FIONBDONE"), exit(1);

    close(fd);
    exit(0);
}
```

For more information on using multiple buffers to speed I/O to raw devices, see `nbuf`(4) and `ioctl`(2) in the *ULTRIX Reference Pages*.

5.4.3.2 Tape Control with ioctl()

The MTIOCTOP `ioctl()` request specifies tape I/O operations. The `mtop` structure, defined in <`mtio.h`>, is used with this request:

```
struct  mtop    {
        short   mt_op;      /* Operation to perform (defined in mtio.h) */
        daddr_t mt_count;   /* Perform it this many times */
};
```

Consider the following example:

```
struct mtop tape_1;

tape_1.mt_op = MTFSF;           /* Move forward one file, */
tape_1.mt_count = 3;            /* three times for each ioctl() call */

ioctl(fd, MTIOCTOP, &tape_1);   /* Move the tape forward three files */
```

Example 5-4 gives an example of tape I/O using `ioctl()`.

See `mtio`(4) in the *ULTRIX Reference Pages* for more information on tape I/O.

Example 5-4: Writing, Rewinding, and Reading a Tape

```
/* tape_read_write.c  Write ten 100-byte records to the /dev/nrmt0h tape.
                      Rewind the tape and read the odd-numbered records.
                      Rewind the tape and read the even-numbered records.
*/
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/mtio.h>
#include <sys/file.h>

#define BUFSIZE 100

main()
{
    struct mtop tape;        /* Tape structure defined in <sys/mtio.h> */
    int fd,                                         /* File descriptor */
        i, j,                                         /* Loop counters */
        status;                                     /* Status variable */
    char buf[BUFSIZE];                                  /* Data buffer */

    fd = open("/dev/nrmt0h",                /* Open the tape device file */
              O_RDWR);                   /* Open for reading and writing */
    if ( fd <= 0 )
         perror("tape_read_write.c: open"), exit(1);

    tape.mt_op = MTREW;                    /* MTREW means rewind the tape */
    tape.mt_count = 1;                                     /* Do it once */

    status = ioctl(fd,                                 /* File to act on */
                   MTIOCTOP,                   /* Perform a tape operation */
                   &tape);/* This structure holds the operation: MTREW */

    if ( status == -1 )                                       /* Failure */
         perror("tape_read_write.c: MTREW"), exit(1);

    for (i = 0 ; i < 10 ; i++)
    {
        for (j = 0; j < BUFSIZE; j++)/* Record 0 contains '0's, record */
            buf[j] = i + 0x30;       /* 1 is '1's, and so on to 9      */

        if ( write(fd, buf, BUFSIZE) == -1 )  /* Write buf to the tape */
             perror("tape_read_write.c: write"), exit(1);     /* Failure */
    }
```

Example 5-4: (continued)

```
    tape.mt_op = MTWEOF;  /* Overwrite MTREW with MTWEOF (write an EOF)*/

    if ( ioctl(fd, MTIOCTOP, &tape) == -1 )    /* Write EOF on the tape */
         perror("tape_read_write.c: MTWEOF"), exit(1);       /* Failure */

    tape.mt_op = MTREW;                 /* Overwrite MTWEOF with MTREW */

    if ( ioctl(fd, MTIOCTOP, &tape) == -1 )          /* Rewind the tape */
         perror("tape_read_write.c: MTREW"), exit(1);        /* Failure */

               /* Read the odd-numbered records and write to terminal */
    tape.mt_op = MTFSR;                     /* Move forward one record */

    while ( read(fd, buf, BUFSIZE) != 0 )             /* Read a record */
    {
        printf("%.100s\n", buf);                     /* Print a record */
        ioctl(fd, MTIOCTOP, &tape);                 /* Skip one record */
    }
    tape.mt_op = MTREW;
    ioctl(fd, MTIOCTOP, &tape);                     /* Rewind the tape */

              /* Read the even-numbered records and write to terminal */
    tape.mt_op = MTFSR;                     /* Move forward one record */
    ioctl(fd, MTIOCTOP, &tape);                     /* Skip one record */

    while ( read(fd, buf, BUFSIZE) != 0 )             /* Read a record */
    {
        printf("%.100s\n", buf);                     /* Print a record */
        ioctl(fd, MTIOCTOP, &tape);                 /* Skip one record */
    }
    close(fd);
}
```

5.4.3.3 Terminal Control with ioctl()

There are several `ioctl()` requests that control terminal I/O operations; all are defined in `<sys/ioctl.h>` and explained in the `tty(4)` reference page. Several different structures are used with these requests because of the patchwork evolution of the ULTRIX terminal driver. Each structure contains a certain part of a terminal's data.

The `sgttyb` structure, defined in `<sgtty.h>`, is used with most of the requests, including TIOCGETP (get terminal characteristics) and TIOCSETP (set terminal characteristics):

```
struct sgttyb {
        char    sg_ispeed;          /* Input speed      */
        char    sg_ospeed;          /* Output speed     */
        char    sg_erase;           /* Erase character  */
        char    sg_kill;            /* Kill character   */
        int     sg_flags;           /* Mode flags       */
};
```

The `sg_flags` element in the `sgttyb` structure can be one of several flags defined in `<sys/ioctl.h>` and explained in the `tty(4)` reference page. Table 5-11 shows a few of the most common flags.

Table 5-11: Common Terminal I/O Modes

Flag	Meaning
RAW	Characters are passed uninterpreted to the program as soon as they are typed. Characters are sent as 8 bits and are not echoed to stdout
CBREAK	Like raw, but characters are echoed, and interrupts, delays, and parity still work
TANDEM	A stop character (DC3) is sent when the input queue is full, and a start character (DC1) is sent when the queue is ready for more input
LCASE	Uppercase characters are converted to lowercase

Example 5-5 shows how to save, change, and restore a terminal's characteristics using `sgttyb`.

Example 5-5: Setting Terminal Characteristics

```
/* sgttyb.c  Set the terminal characteristics (in the sgttyb structure)
             to CBREAK and NOECHO.  Read characters and echo
             incorrectly at the terminal until CTRL/A is typed.
*/
#include <stdio.h>
#include <sgtty.h> /*Includes <sys/ioctl>, which includes <sys/ttyio>*/
#include <sys/file.h>

main()
{
    struct sgttyb orig_settings, /*Structure to hold current settings*/
                  new_settings;      /*Structure to hold new settings*/
    int fd,                                      /*File descriptor*/
        status;                                  /*Status variable*/
    char c;                         /*Character read from the terminal*/

                  /*Because stdin or stdout could be redirected, open*/
                  /*"/dev/tty", which is guaranteed to be my terminal*/
    if ( (fd = open("/dev/tty", O_RDWR,         /*Open for read-write*/
                    0)) == -1 )
        perror("sgttyb.c: open"), exit(1);                  /*Failure*/

    status = ioctl(fd,                      /*For this terminal       */
                   TIOCGETP,                /*get the current settings*/
                   &orig_settings);         /*and store them here     */
    if ( status == -1 )
         perror("sgttyb.c: ioctl GET 1"), exit(1);          /*Failure*/

    status = ioctl(fd, TIOCGETP, &new_settings);/*Store them here too*/
    if ( status == -1 )
         perror("sgttyb.c: ioctl GET 2"), exit(1);          /*Failure*/

    new_settings.sg_flags &= ~ECHO;                /*Turn off echoing*/
    new_settings.sg_flags |= CBREAK;             /*Turn on CBREAK mode*/

    if ( ioctl(fd, TIOCSETP, &new_settings)    /*Install new settings*/
         == -1 )
         perror("sgttyb.c: ioctl SET 1"), exit(1);
```

Example 5-5: (continued)

```
    write(fd, "Type some character\n", 21);         /*Solicit input*/
    read(fd, &c, 1);                              /*Read a character*/
    while (c++ != '\001')     /*Increment c, and while not CTRL/A...*/
    {
        write(fd, &c, 1);                           /*Write the next*/
        read (fd, &c, 1);                         /*Read a character*/
    }
    if ( ioctl(fd, TIOCSETP, &orig_settings)    /*Restore the original*/
         == -1 )                                /*terminal settings   */
         perror("sgttyb.c: ioctl SET 2"), exit(1);

    exit(0);
}
```

The `tchars` structure, defined in <`sys/ttyio.h`>, is used with the TIOCGETC (get special characters) and TIOCSETC (set special characters) requests:

```
struct tchars {
        char    t_intrc;    /* Interrupt                Default = CTRL/? */
        char    t_quitc;    /* Quit                     Default = CTRL/\ */
        char    t_startc;   /* Start output             Default = CTRL/Q */
        char    t_stopc;    /* Stop output              Default = CTRL/S */
        char    t_eofc;     /* End-of-file (EOF)        Default = CTRL/D */
        char    t_brkc;     /* Input delimiter (like nl)    Default = -1 */
};
```

Example 5-6 shows how to store, change, and reset a terminal's special characters using `tchars`.

Example 5-6: Changing a Terminal's Special Characters

```
/* tchars.c  Change the EOF character to be CTRL/A
*/
#include <sgtty.h>
#include <stdio.h>

main()
{
    struct tchars orig_char,                   /* The original settings */
                  new_char;                     /* The altered settings */

    ioctl(0,                                   /* Descriptor 0 is stdin */
          TIOCGETC,                       /* Get the special characters */
          &orig_char);                             /* Store them here */

    ioctl(0, TIOCGETC, &new_char);              /* Store them here also */

    new_char.t_eofc = '\001';                       /* Change EOF to ^A */
    ioctl(0, TIOCSETC, &new_char);           /* Reset the special chars */

    puts("Executing cat > myfile  CTRL/A is EOF"); /* Prompt for input */
    system("cat > myfile");                     /* Execute cat command */

    ioctl(0, TIOCSETC, &orig_char);     /* Reset to the original chars */

    exit(0);
}
```

The `ltchars` structure, defined in <`sys/ttyio.h`>, is used with the TIOCGLTC (get local special characters) and TIOCSLTC (set local special characters) requests:

```
struct ltchars {
       char    t_suspc;   /* Stop process signal       Default = CTRL/Z */
       char    t_dsuspc;  /* Delayed stop process      Default = CTRL/Y */
       char    t_rprntc;  /* Reprint line              Default = CTRL/R */
       char    t_flushc;  /* Flush output (toggles)    Default = CTRL/O */
       char    t_werasc;  /* Word erase                Default = CTRL/W */
       char    t_lnextc;  /* Literal next character    Default = CTRL/V */
};
```

The remaining structure is just a word, defined in `<ioctl.h>` and explained in the `tty`(4) reference page, that is used with the four local-mode word requests, such as TIOCLGET (get the current local word) and TIOCLBIC (clear these bits in the local word). The local-mode word consists of several values (that can be combined using a bitwise OR); for example:

- LTOSTOP—Send SIGTTOU for background output
- LTILDE—Convert ~ to ‘ on output (for Hazeltine terminals)
- LCTLECH—Echo input control characters as ^X, and delete as ^?

Example 5-7 shows how to change a terminal's local mode word.

Example 5-7: Changing a Terminal's Local Mode Word

```
/* local_word.c  Turn off echoing of control characters
                 in the ^<character> form
*/
#include <stdio.h>
#include <sgtty.h>

main()
{
    short word;                        /* Holds the terminal's local word */
    int status;                                     /* Returned status */

    status = ioctl(0,                            /* Descriptor 0 is stdin */
                   TIOCLGET,                      /* Get the current word */
                   &word);                               /* Store it here */
    if ( status == -1 )
         perror("local_word.c: ioctl GET"), exit(1);           /* Failure */

    word &= ~LCTLECH;              /* Change the appropriate bits in word */

    if ( ioctl(0, TIOCLSET, &word) == -1)/* Set the local word to word */
         perror("local_word.c: ioctl SET"), exit(1);           /* Failure */

    exit(0);
}
```

Interprocess Communication 6

This chapter describes three different methods of interprocess communication that can be used in programs:

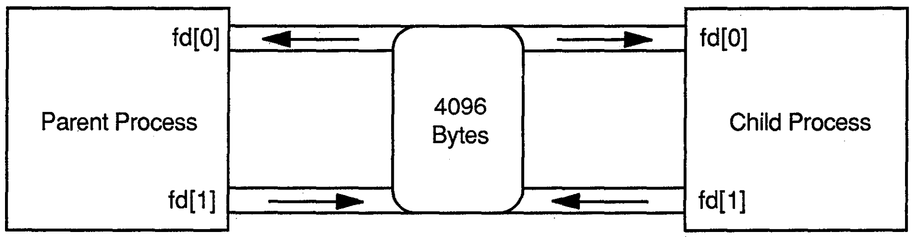
- Pipes
- Signals
- Sockets

6.1 Pipes

A pipe is a memory buffer. Each pipe holds up to PIPE_MAX bytes of data (PIPE_MAX, defined in `<limits.h>`, is usually 4096). Pipes are created by the `pipe()` system call, and are accessed by file descriptors contained in an integer array.

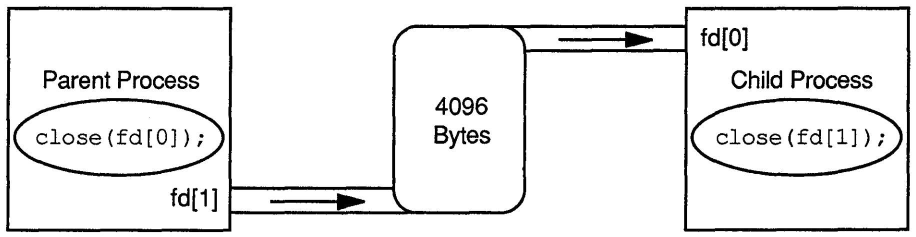
Pipes can only be used between related processes: between parent and child, or between siblings (child processes from the same parent). Figure 6-1 depicts a parent process and child process communicating through a pipe.

Figure 6-1: A Pipe



As Figure 6-1 shows, both processes can read to and write from the pipe, and each process can read the data it has written. Therefore, it is prudent to use pipes for one-way communication by closing (with the `close()` system call) either the write or read end of the pipe in each process, as shown in Figure 6-2.

Figure 6-2: A One-Way Pipe



The program in Example 6-1 creates a pipe, creates a child process, and then communicates with the child, as depicted in Figure 6-2. A child process has access to pipes created by its parent before the child was created.

Example 6-1: Creating a Child and a Pipe

```
/* pipe.c       Creates a pipe and a child process.  The parent reads a
                line from stdin and writes it to the pipe.  The child
                reads a line from the pipe and writes it to stdout.
*/

#include <stdio.h>

main()
{
int  pid,            /* Process ID returned by fork()                   */
     n,              /* Number of bytes read from the pipe by the child */
     fd[2];          /* Array that holds the pipe file descriptors      */
char par_line[81],   /* Line buffer for parent                          */
     chi_line[81];   /* Line buffer for child                           */

    if ( pipe(fd) == -1 )                             /* Create a pipe */
        perror("pipe.c: pipe failed"), exit(1);

    if ( (pid = fork()) == -1 )                      /* Create a child */
        perror("pipe.c: fork failed"), exit(1);

    if (pid == 0)               /* Child process; execute child's code */
    {
        close(fd[1]);                       /* Close write end of pipe */
        n = read(fd[0], chi_line, 80);                /* Read from pipe */
        chi_line[n] = '\0';
        printf("Child: your line was %s\n", chi_line);
        exit(0);                        /* Successful exit from child */
    }

    else                      /* Parent process; execute parent's code */
    {
        close(fd[0]);                       /* Close read side of pipe */
        printf("Enter line: ");
        gets(par_line);                      /* Read a line from stdin */
        write(fd[1], par_line, strlen(par_line));/* Write line to pipe */
        wait(0);                             /* Wait for child to exit */
        exit(0);                       /* Successful exit from parent */
    }
}
```

The following list contains information about pipes:

- Processes communicating through pipes must be related: parent and child, or siblings.
- By convention, always read file descriptor `[0]`, and write file descriptor `[1]`, in both parent and child.
- Messages in pipes have no record boundaries; for example, a parent could write 100 bytes, and a child could read 25 bytes 4 times, or the other way around.
- A process can read its own data from a pipe.
- Reading an empty pipe blocks the process; the process waits until there is something in the pipe to read, unless specific measures are taken. (See Section 6.3.4.)
- Writing to a full pipe (4096 bytes on most systems) blocks the process; the process waits until the pipe empties enough to take the message, unless specific measures are taken. (See Section 6.3.4.)
- If all write channels (`fd[1]`) to a pipe are closed, the reader of that pipe will read an EOF when that pipe is empty.
- Reads from a pipe destroy the data; the data in a pipe cannot be peeked at.

6.1.1 Redirecting stdin, stdout, and stderr to Pipes

ULTRIX provides up to 64 file descriptors per process (numbered 0-63), which the system uses as handles on various objects: disk files, special files, sockets, pipes, and others. The first three file descriptors in any process are:

- File descriptor 0, Standard input (stdin)
- File descriptor 1, Standard output (stdout)
- File descriptor 2, Standard error (stderr)

Subsequent file descriptors are allocated sequentially; a `pipe()` call, for example, returns file descriptor 3 for reading and 4 for writing. (In Example 6-1, these values are stored in `fd[0]` and `fd[1]`, respectively.)

The `dup()` system call allocates a new file descriptor that points to an object already pointed to by a file descriptor. Therefore, `dup()` can be used to redirect a process's stdin, stdout, or stderr to a pipe.

The idea is to create a pipe, then close (with the `close()` system call) an existing file descriptor, stdout for example, and then immediately call `dup()`, supplying the write channel to the pipe as the object to which the newly allocated descriptor points. Because `dup()` always allocates the lowest available file descriptor, it will allocate file descriptor 1, the descriptor given up when stdout was closed. Now, any writes to stdout (which the system knows as file descriptor 1) are written to the pipe. The writer, for example the `ls` shell command, writes to file descriptor 1 just as it always does, but now file descriptor 1 points to a pipe rather than stdout. Figure 6-3, Figure 6-4, and Figure 6-5 depict what happens.

Figure 6-3 shows the file descriptors that are allocated for a process and its child after a pipe has been created. The numbers in Figure 6-3 are the file descriptors for stdin, stdout, and stderr. The values of `fd[0]` and `fd[1]` are 3 and 4, respectively.

Figure 6-3: File Descriptors of Two Processes with a Pipe

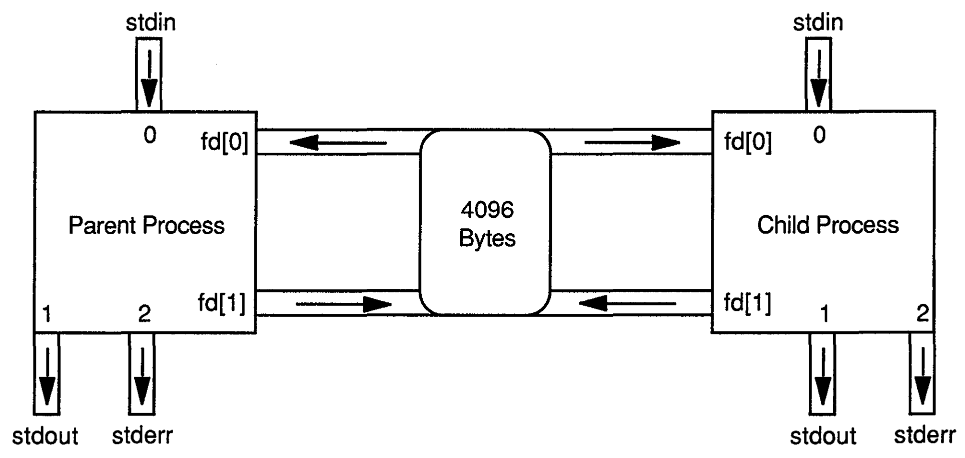


Figure 6-4 depicts the processes in Figure 6-3 after the parent executes the following system calls:

```
close(fd[0]);   /* Close the pipe read descriptor                     */
close(1);       /* Close stdout                                       */
dup(fd[1]);     /* Allocate the lowest available descriptor, which    */
                /*   is the just-closed 1, and make it point to what */
                /*   fd [1] points to (the pipe write descriptor)    */
close(fd[1]);   /* Close the original pipe write descriptor           */
```

Figure 6-4: stdout Redirected in a Parent

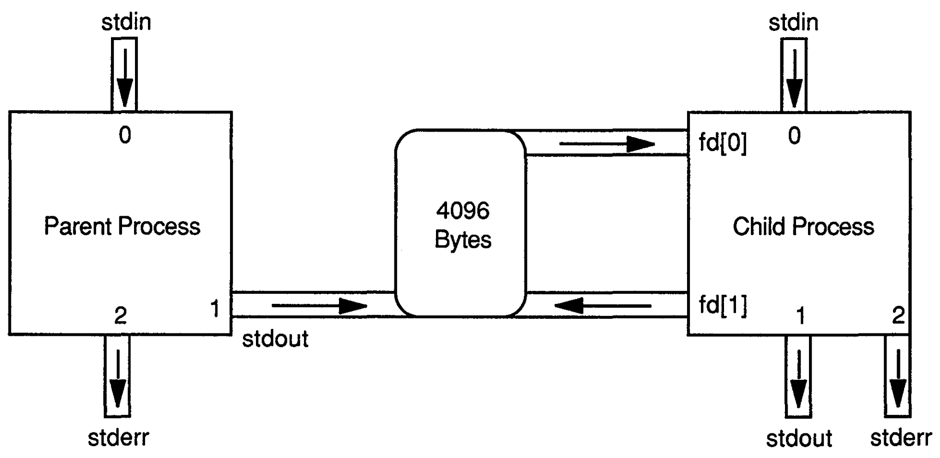
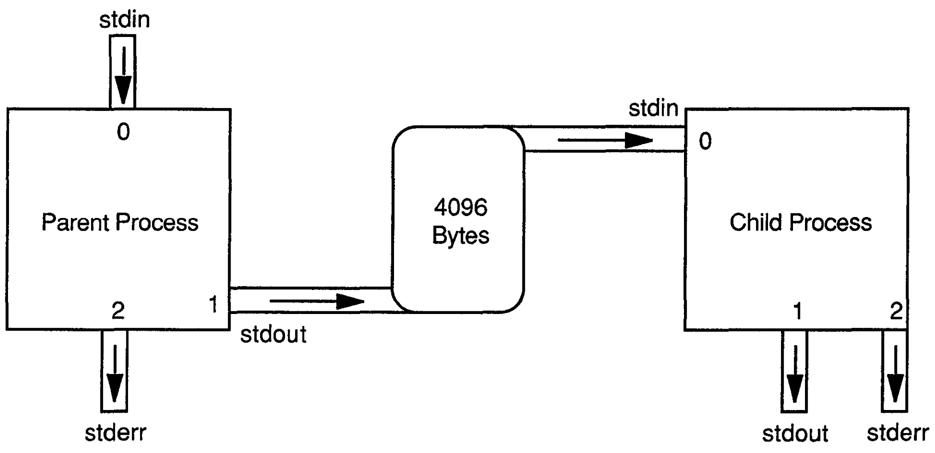


Figure 6-5 depicts the processes in Figure 6-4 after the child executes the following system calls:

```
close(fd[1]);  /* Close the pipe write descriptor                 */
close(0);      /* Close stdin                                     */
dup(fd[0]);    /* Allocate the lowest available descriptor, which  */
               /*   is the just-closed 0, and make it point to what */
               /*   fd[0] points to (the pipe read descriptor)     */
close(fd[0]);  /* Close the original pipe read descriptor         */
```

Figure 6-5: stdout Redirected in a Parent, and stdin Redirected in a Child



Now, all four of the original file descriptors into and out of the pipe are closed. The parent's stdout writes to the pipe, and the child's stdin reads from the pipe.

The program in Example 6-2 establishes the communication channels shown in Figure 6-5, then executes a shell command that writes to stdout (`who`) in the parent, and executes a shell command that reads from stdin (`wc -l`) in the child.

Example 6-2: Redirecting stdin and stdout to a Pipe

```
/* redirect.c       This program uses the dup() system call to implement
                    this shell command: who | wc -l, which tells how many
                    users are on a system.
*/

#include <stdio.h>

main()
{
int pid,                    /* Process ID returned by fork()            */
    fd[2];                  /* Array that holds the pipe file descriptors */

    if ( pipe(fd) == -1 )                               /* Create a pipe */
        perror("redirect.c: pipe failed"), exit(1);

    if ( (pid = fork()) == -1 )                        /* Create a child */
        perror("redirect.c: fork failed"), exit(1);

    if ( pid == 0 )              /* Child process; execute child's code */
    {
              /* Make stdin the read channel of the pipe and exec 'wc' */
        close(fd[1]);                         /* Close write side of pipe */
        close(0);                      /* Close stdin (file descriptor 2) */
        dup(fd[0]);     /* Make file descriptor 2 (stdin) same as fd[0] */
        close(fd[0]);                        /* Close old read end of pipe */

        if ( execlp("wc", "wc", "-l", 0) == -1)           /* Run wc -l */
            perror("redirect.c child: execl failed"), exit(1);
    }

    else                        /* Parent process: execute parent's code */
    {
           /* Make stdout the write channel of the pipe and exec 'who' */
        close(fd[0]);                          /* Close read side of pipe */
        close(1);                     /* Close stdout (file descriptor 1) */
        dup(fd[1]);     /* Make file descriptor 1 (stdout) same as fd[1] */
        close(fd[1]);                       /* Close old write end of pipe */

        if ( execlp("who", "who", 0) == -1 )                  /* Run who */
            perror("redirect.c parent: execlp failed"), exit(1);
    }
}
```

6.1.2 Creating Pipes with popen()

Use the `popen()` library routine to create a child process to execute a Bourne shell (`sh`) command. A `popen()` call also creates a one-way pipe between the parent and child processes. The `popen()` routine is like a combination of `pipe()`, `fork()`, and `exec()`. Here is what the `popen()` library routine does:

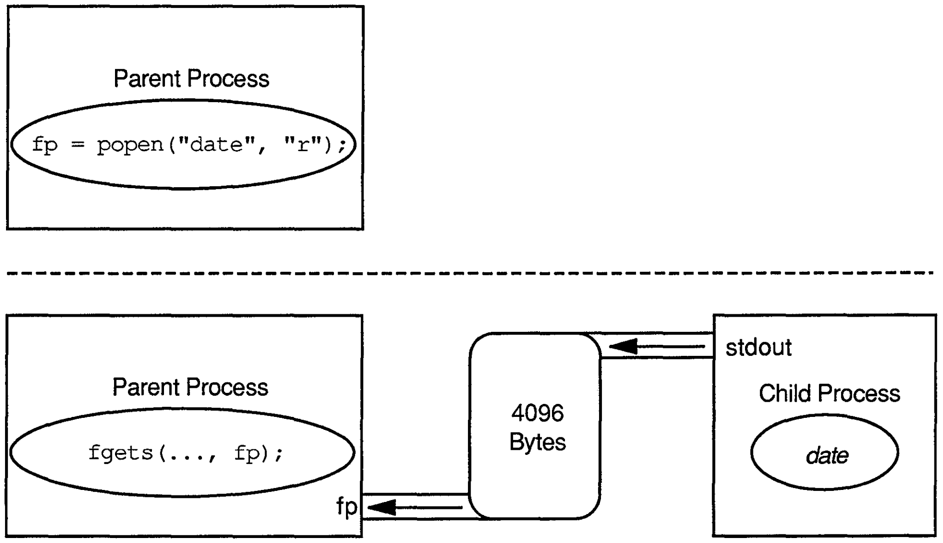
1. Creates a pipe
2. Creates a child process
3. Creates a Bourne shell in the child process that executes the shell command specified in the `popen()` call
4. Causes the shell command to read or write the pipe to communicate with the parent process

5. Returns a standard I/O file pointer as the channel to the pipe for the parent to read or write

Figure 6-6 illustrates what happens when the following `popen()` call is made:

```
#include <stdio.h>
FILE *fp;
fp = popen("date", "r");    /* The parent can read the output of date */
    .                       /* using the standard I/O file pointer fp */
    .
    .
exit_status = pclose(fp);   /* Close the file pointer */
```

Figure 6-6: Calling popen() with date for Reading



The program in Example 6-3, `who_pipe.c`, does what the program in Example 6-2 does; it implements the shell command `who | wc -l`. But `who_pipe.c` uses `popen()` rather than `dup()`. Figure 6-7 depicts what happens when `who_pipe` is run.

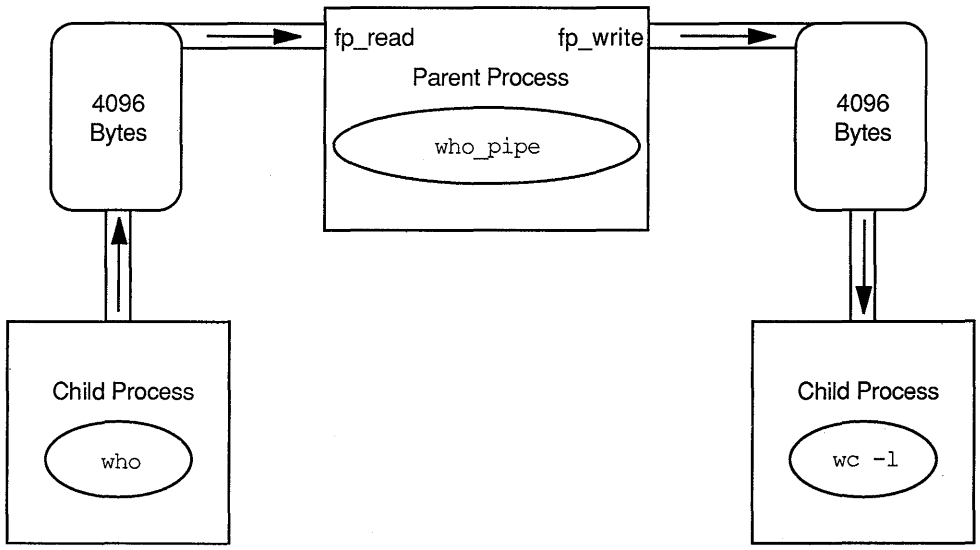
Example 6-3: Creating Child Processes to Run Shell Commands

```
/* who_pipe.c       Create two child processes to implement the shell
                    command: who | wc -l
*/

#include <stdio.h>
#include <ctype.h>

main()
{
    char buf[133];      /* A line buffer                              */
    FILE *fp_read,      /* A file pointer that reads from the first pipe */
         *fp_write;     /* A file pointer that writes to the second pipe */

    if ( (fp_read = popen("who", "r")) == NULL ) /* Create a process    */
```

Example 6-3: (continued)

```
        perror("who_pipe: popen who"), exit(1);  /* running 'who' that */
                                                 /* can be read from   */
                                                 /* fp_read            */

    if ( (fp_write = popen("wc -l", "w"))        /* Create a process   */
          == NULL )                              /* running 'wc' that  */
        perror("who_pipe: popen wc"), exit(1);   /* can be written to  */
                                                 /* through fp_write   */

    while ( fgets(buf, 132, fp_read) != NULL )   /* Read a line from   */
        fputs(buf, fp_write);                    /* 'who' (fp_read)    */
                                                 /* and write it to    */
                                                 /* 'wc' (fp_write)    */

    pclose(fp_read);                         /* Close the file pointer */
    pclose(fp_write);                        /* Close the file pointer */

    exit(0);                                       /* Successful exit */
}
```

Figure 6-7: who_pipe.c



6.2 Signals

ULTRIX defines a set of about 30 signals that can be delivered to a process. If a signal is not caught by the receiving process, that process is subject to the default behavior for processes receiving that signal, which can be termination, termination with a core image produced, or no action at all.

Signals are caught by the `sigvec()` system call and the `signal()` library routine. This section discusses only `signal()` because it is an easier-to-use version of `sigvec()`.

A process that uses `signal()` can take one of three actions for each type of signal it receives:

- Ignore the signal. The signal is discarded as if it were never sent.
- Block the signal. The signal is queued but not delivered.
- Catch the signal. Control is passed to a routine written by the user to handle a particular signal.

To see all the signals available on your system, see `sigvec`(2) in the *ULTRIX Reference Pages*. The following signals are some of the more common ones:

- SIGINT, Interrupt (can be generated from a keyboard by Ctrl/C)
- SIGQUIT, Quit
- SIGKILL, Kill (cannot be caught, blocked, or ignored)
- SIGSYS, Bad argument to system call
- SIGSTOP, Stop (cannot be caught, blocked, or ignored)
- SIGPIPE, Write on a pipe with no reader
- SIGALRM, Alarm clock
- SIGFPE, Floating point exception
- SIGIO, I/O has become possible on a descriptor
- SIGUSR1, User-defined signal 1
- SIGUSR2, User-defined signal 2

Signals can only be sent between the following types of processes:

- From the kernel to a process
- Between parent and child processes
- Between unrelated processes that have the same UID

6.2.1 Catching Signals

The `signal()` library routine can be used to catch signals. A process can catch many different signals and, except in the System V environment, once `signal()` is called for a particular signal type, that signal is caught each time it is received. (In the System V environment, it is caught only once; `signal()` must be recalled to catch the signal again.)

The `signal()` routine takes two arguments. The first argument is the signal to catch, such as SIGINT. The second argument is what to do with the signal. The second argument can be the address of a signal handler function, or one of these values defined in `<signal.h>`:

- SIG_IGN, ignore the signal
- SIG_DFL, accept the default action for the signal, useful for resetting signal behavior
- SIG_ERR, terminate the process

The `signal()` routine returns the previous action for the signal. If the following call to `signal()` is the first for SIGINT in a program, the value of `x` is SIG_DFL (default action), unless changed by the shell (see Section 6.2.2):

```
x = signal(SIGINT, SIG_IGN);
```

If a signal handler had been previously declared for SIGINT, the address of that handler would have been returned.

The program in Example 6-4 ignores interrupts. If run, its infinite loop must be terminated some other way.

Example 6-4: Ignoring a Signal

```
/* signal.c      Catch the SIGINT (interrupt) signal and ignore it.
*/

#include <signal.h>

main()
{
    signal(SIGINT, SIG_IGN); /* Ctrl/C will not stop this program, but */
                             /* Ctrl/Z or kill -9 will.                */

    for(;;)
        printf("Looping forever.  Ctrl/C is useless against me.\n");
}
```

6.2.2 Handling Signals

The second argument to `signal()` can be the address of a signal handler. A routine that is declared to be a signal handler receives three arguments when the signal it handles is received by the process; it need not use any of them. The three parameters have the following syntax:

```
signal_handler(int signal_number; long code; struct sigcontext *scp)
```

- `signal_number` is the value of the signal as defined in <`signal.h`>.
- `code` is an additional piece of information, usually supplied with SIGFPE (floating point exception), that further specifies the cause of the signal; for example:
  - FPE_INTOVF_TRAP, for integer overflow
  - FPE_FLTDIV_TRAP, for floating-point division by zero
  - ILL_RESAD_FAULT, for attempting to access a reserved address

  All codes are listed in <`signal.h`>.
- `scp` points to a structure of type `sigcontext`, defined in <`signal.h`>. The `sigcontext` structure stores the process context before a signal was sent, in the event that it needs to be restored after receiving a signal.

The program in Example 6-5 does not ignore the SIGINT signal; it handles it. SIGINT does stop `write_text.c`, but not before the program cleans up after itself.

Example 6-5: Handling a Signal

```
/* write_text.c      Prompt for input, place input in file 'tmp'.
                     If interrupted by CTRL/C, remove 'tmp' and exit.
*/

#include <stdio.h>
#include <signal.h>

main()
{
    FILE *fp;                          /* File pointer to 'tmp'        */
    char c;                            /* Character read from terminal */
    void sigint_handler();             /* The SIGINT signal handler    */

    if ( signal(SIGINT, SIG_IGN)         /* If SIGINT is already being */
         != SIG_IGN )                    /* ignored, don't declare a   */
                                         /* handler for it (see text)  */
        signal(SIGINT,               /* Make sigint_handler handle all    */
               sigint_handler);      /* SIGINT signals.  signal() blocks */
                                     /* other SIGINTs while a SIGINT is   */
                                     /* being handled                     */

    fp = fopen("tmp", "w");             /* Open file 'tmp' for writing */
    printf("Enter text.\n");                        /* Prompt for text */
    while ( (c=getchar()) != EOF)    /* Get a char and write it to 'tmp' */
        putc(c, fp);

    puts("EOF typed before CTRL/C");
    exit(0);                                        /* Successful exit */
}

void sigint_handler()           /* Remove the file 'tmp', and kill this */
{                               /* program.  Do not return to main()    */
    if ( unlink("tmp") != -1)
        puts("The tmp file has been removed.");
    exit(1);
}
```

A signal sent from the keyboard, such as an interrupt (SIGINT), is sent to all processes associated with that terminal. However, the shell turns off interrupts sent to background processes (those started with an ampersand [&] at the end of their command line). That is why the program in Example 6-5 called `signal()` for SIGINT and tested its value (see Section 6.2.1) before declaring a handler for SIGINT. If `write_text.c` declares that all SIGINTs are to be handled by its handler, then the shell does not turn off interrupts when the process is run in the background. The `write_text.c` program tests the current state of interrupt handling, and continues to ignore interrupts if they are currently being ignored.

If you want your program to detect and handle signals, but your program cannot be stopped just anywhere, have your signal handler merely set a flag and return. Execution resumes at the exact point the signal occurred. The flag can be tested after the crucial code path is complete. A similar strategy is signal blocking (see Section 6.2.4).

The program in Example 6-6 shows how to handle arithmetic exceptions and take an action based on the type of exception, as revealed in the `code` argument passed to the signal handler. The `arith_trap.c` program also shows a difference between faults and traps. With traps, the PC is incremented before a signal is handled; when execution returns from the signal handler, the next instruction is executed. With faults, the PC is not incremented; when execution returns from the signal handler, the faulting instruction is executed again. Therefore, the signal handler in `arith_trap.c` exits after a fault.

Example 6-6: Using a Signal Handler's code Argument

```
/* arith_trap.c     Establish a signal handler for arithmetic traps.
                    Create a division by zero trap and a floating-point
                    overflow fault to try the mechanism.  (Note that this
                    program will not compile using c89 because the c89
                    compiler detects the attempt to divide by zero.)
*/

#include <signal.h>
#include <stdio.h>

main()
{
    void sigfpe_handler();     /* The signal handler                    */
    short i, j;                /* Variables used in causing the faults, */
    float r, s;                /* thereby generating SIGFPE signals     */

    signal(SIGFPE, sigfpe_handler);  /* Make the SIGFPE signal handler */

    j = 0;
    i = 32 / j;               /* Cause a divide by zero arithmetic trap */

    r = 1.0e20;
    s = r * r;                 /* Cause a floating-point overflow fault */

    exit(0);                                        /* Successful exit */
}

void sigfpe_handler(signal_number, code)
     int signal_number, code;
{
    printf("Signal %d received\t", signal_number);

    switch (code)
    {
        case FPE_INTOVF_TRAP : puts("Integer Overflow");
                               break;

        case FPE_INTDIV_TRAP : puts("Integer Division by Zero");
                               break;

        case FPE_FLTOVF_FAULT: puts("Floating Overflow Fault");
                               exit(1);     /* Because the PC points to */
                                            /* the faulting instruction */
        default:
        printf("Code = %d\n", code);
        exit(1);
    }
}
```

6.2.3 Sending Signals

Signals can be sent from a keyboard. To see which signals are mapped to keys on your keyboard, issue the command `stty everything`. Signals sent from a keyboard are received by all processes in the process group associated with the terminal.

Signals can be sent between related processes: parent and child, and siblings. A Process sends a signal to another process by using the `kill()` system call, which takes the process ID of the receiving process as its first argument, and the signal to be sent (such as SIGPIPE) as its second argument.

A process sends a signal to a process group by using the `killpg()` system call, which is the same as `kill()` except that the first argument is the process group ID. (Process group IDs are returned by the `getpgrp()` system call.)

The program in Example 6-7, has a parent process send a signal to its child, which handles the signal and exits.

Example 6-7: Sending a Signal Between Processes

```
/* signal_child.c       Parent process sends SIGINT to a child process.
                        The child process handles the signal and exits.
*/

#include <signal.h>
#include <stdio.h>

main()
{
    int pid;              /* The child's process ID returned by fork() */
    void SIGINT_handler();              /* The signal handler routine */

    if ( (pid = fork()) == 0 )  /* Child process; execute child's code */
    {
        signal(SIGINT, SIGINT_handler);         /* Make signal handler */
        pause();                                  /* Wait for a signal */
    }

    else                      /* Parent process; execute parent's code */
    {
        sleep(1);                /* Wait a second for child to be born */
        kill(pid, SIGINT);                      /* Send signal to child */
        wait(0);                         /* Wait until child terminates */
        exit(0);                                     /* Successful exit */
    }
}

void SIGINT_handler(signal_number)     /* Identify the signal received */
     int signal_number;                        /* (SIGINT = 2) and exit */
{
    printf("Signal %d received from parent.\n", signal_number);
    exit(0);                                         /* Successful exit */
}
```

6.2.4 Blocking Signals

A signal can be blocked to protect certain sections of code from receiving signals when the work being done should not be interrupted. Unlike ignoring a signal, blocking a signal merely postpones it until the process is ready to handle the signal, after the crucial code section has been executed.

A blocked signal is put in a queue and handled as soon as the block is released. The order in which the blocked signals are released is implementation dependent. Multiple occurrences of the same signal are not saved.

The `sigblock()` system call blocks signals through the use of a signal mask; if the *n*th bit in the mask is set, signal *n* is blocked. (See `<signal.h>` for the values of signals.) After the crucial code has been executed, the `sigpause()` system call is used to release any blocked signals from the queue, and restore the old mask:

```
long oldmask;

oldmask = sigblock(0);                          /* Get the current mask */
sigblock(SIGSYS | SIGTRAP);                  /* Block SIGSYS and SIGTRAP */

/* Code protected from SIGSYS and SIGTRAP goes here */

sigpause(oldmask);     /* Release blocked signals and restore old mask */
```

6.2.5 Signals and Timers

The `alarm()` library routine sends the SIGALRM signal to the calling process. SIGALRM terminates the process if it is not caught or blocked. The program in Example 6-8 shows how use `alarm()`.

Example 6-8: Using alarm()

```
/* alarm.c      Use alarm() to send SIGALRM in the number
                of seconds specified on the command line
*/

#include <signal.h>
#include <stdio.h>

main(argc,argv)
char **argv;
int argc;
{
    void sigalrm_handler();
    char *strcpy();

    if (argc != 2) {
        fprintf(stderr, "Usage: %s seconds\n", argv[0]);
        exit(1);
    }

    signal(SIGALRM, sigalrm_handler);          /* Make SIGALRM handler */

    alarm((unsigned) atoi(argv[1]));  /* Make argv[1] an unsigned int, */
                                /* and send SIGALRM in argv[1] seconds */

    pause();                    /* Block until the signal is delivered */

    printf("Back in main().\n");
}

void sigalrm_handler()                               /* SIGALRM handler */
{
    printf("Awake after alarm.\n");
}
```

6.2.5.1 Timed Intervals

For each process, the system provides three timers that take a starting value and count down to zero:

- ITIMER_REAL, which counts down in real time. A SIGALRM signal is sent when ITIMER_REAL expires.
- ITIMER_VIRTUAL, which counts down in process virtual time (runs only while the process has the CPU in nonsystem mode, also known as user mode). A SIGVTALRM is sent when ITIMER_VIRTUAL expires.
- ITIMER_PROF, which counts down in both process virtual time and when the system is running on behalf of the process. This timer is meant to be used for code profiling. A SIGPROF signal is sent when ITIMER_PROF expires.

These timers are set with the `setitimer()` system call, which takes two arguments, and an optional third argument. The first argument is the name of the timer to be set (such as ITIMER_REAL). The second argument is the address of an `itimerval` structure, defined in <`time.h`>, which contains the amount of time to count down for the first iteration, and the amount of time to count down for subsequent iterations. The itimerval structure is shown in Example 6-9.

Example 6-9: The itimerval Structure

```
struct  itimerval {
        struct  timeval it_interval;    /* Time to count down
                                           after the first */
        struct  timeval it_value;       /* Time to count down
                                           the first time */
};

struct  timeval {
        long    tv_sec;                 /* Seconds */
        long    tv_usec;                /* Microseconds */
};
```

If `itimerval.it_interval` is zero, the timer counts down once; if it is non-zero, the timer repeats for the life of the process. A signal (SIGALRM, SIGVTALRM, or SIGPROF, depending on the timer used) is sent after `it_value` time elapses, and a signal is sent thereafter each time `it_interval` time elapses.

The optional third argument is also the address of an `itimerval` structure; this one is used for receiving the previous values contained in `itimerval`.

The `getitimer()` system call gets the current values in the `itimerval` structure for the specified timer. The program in Example 6-10 shows how `setitimer` and `getitimer` can be used.

Example 6-10: Using setitimer and getitimer

```
/* timer.c      Illustrate the use of the setitimer() and getitimer()
                system calls.
*/

#include <signal.h>
#include <time.h>
#include <stdio.h>

main()
{
    short i, j;                     /* Loop counter and generic variable */
    void announce();                         /* SIGALRM's signal handler */
    struct itimerval val, current_val;            /* Defined in <time.h> */

    signal(SIGALRM, announce);                    /* Handler for SIGALRM */

    val.it_interval.tv_sec = 3;  /* Count 3 sec. each time after first */
    val.it_interval.tv_usec = 0;                          /* Microseconds */
    val.it_value.tv_sec = 10;      /* Count 10 seconds the first time */
    val.it_value.tv_usec = 0;                             /* Microseconds */

    setitimer(ITIMER_REAL, &val, 0);                   /* Start the timer */

    for(i = 0 ; i < 100000; i++)
    {
    j = (i*34987 + 89320.5)/41;                          /* Waste time */

    if ( (i % 25000) == 0 ) /* Occasionally show how much time remains */
        {
        getitimer(ITIMER_REAL, &current_val);
        printf("it_value: %d\n", current_val.it_value.tv_sec);
        printf("it_interval: %d\n", current_val.it_interval.tv_sec);
        }
    }
}

void announce()
{
    puts("\nTimer expired\n");
}
```

6.3 Sockets

Sockets are another interprocess communication mechanism. Sockets are similar to pipes, except the communicating processes need not be related, and there is only a single channel for each process, but this channel is full duplex (two-way read and write).

Sockets are created with a domain. The domains supported on your system can be found in <socket.h>. Here are a few common domains:

- AF_UNIX, for processes on the same node
- AF_INET, (internet) for processes on different nodes using TCP/IP
- AF_DECnet, for processes on different nodes using DECnet
- AF_SNA, for processes on different nodes using SNA

Each socket has a certain type, which defines its communication semantics. The following list shows the most common types and their attributes (see `<socket.h>` for all types on your system):

- Stream: SOCK_STREAM
  - Available in all domains
  - A connected socket
  - Data is sequenced, reliable, and unduplicated
  - Record boundaries are not preserved (for example, 10 bytes can be written three times, then all 30 bytes can be read at once)
- Datagram: SOCK_DGRAM
  - Available only in AF_UNIX and AF_INET domains
  - An unconnected socket
  - Data is not guaranteed to be sequenced, reliable, or unduplicated (user protocols must be used)
  - Record boundaries are preserved
- Sequenced Packet: SOCK_SEQPACKET
  - Available only in AF_DECnet domain
  - Same as stream, but record boundaries are preserved
- Raw: SOCK_RAW
  - No specific semantics
  - Used for unprocessed access to internal network layers

If a process reads a datagram socket immediately after writing to it, it can read back its own message. This is not true of stream sockets.

For each socket in a program, there must be a structure that receives the socket name. Each structure is defined in a header, as shown in Table 6-1.

Table 6-1: Socket Name Structures

Domain	Structure	Header
AF_UNIX	`sockaddr_un`	`<un.h>`
AF_INET	`sockaddr_in`	`<netinet/in.h>`
AF_DECnet	`sockaddr_dn`	`<netdnet/dn.h>`

6.3.1 Using a Datagram Socket Between Processes on the Same Node

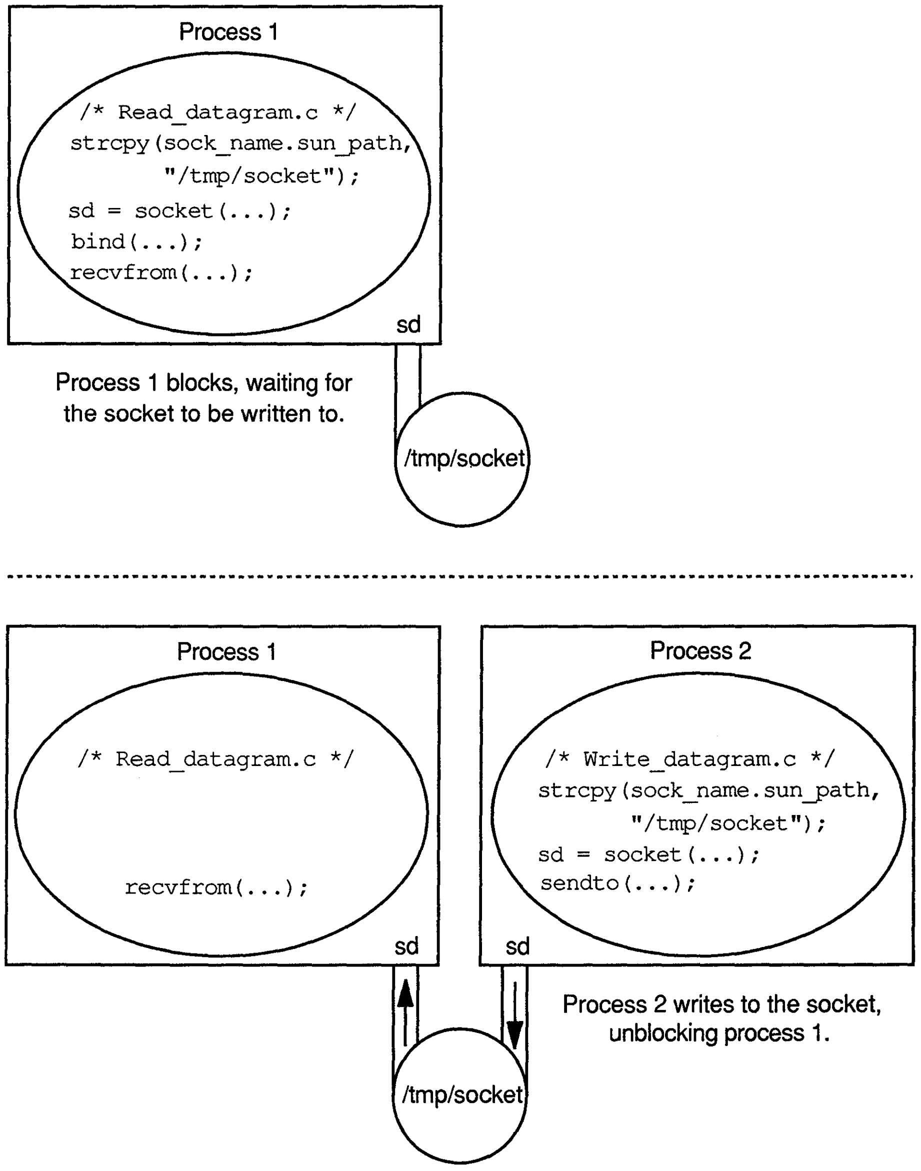
The two example programs in this section communicate through a datagram socket in the AF_UNIX domain (which means both processes must be on the same node). Both programs must obtain a socket descriptor and close the socket when they are through, but only one binds a name to the socket and removes the socket file after closing the socket, as shown in Table 6-2.

Table 6-2: System Calls for Datagram Socket Communication

Server Program		Client Program
`sd = socket();`	Get a socket descriptor (like a file descriptor).	`sd = socket();`
`bind();`	Name the socket. The name is used by both processes to communicate through the same socket.	
`sendto(); recvfrom();`	Send and receive messages	`sendto(); recvfrom();`
`close();`	Close the socket.	`close();`
`unlink();`	Remove the socket file (AF_UNIX domain only). If the file is not removed, future attempts to bind to a socket of the same name will fail.	

The program in Example 6-11 creates a datagram socket and reads from it. The program in Example 6-12 writes to that socket. Figure 6-8 illustrates how these two independent processes communicate. Their only connection is that both use the same socket name.

Figure 6-8: Communicating Through a Datagram Socket



Example 6-11: Creating and Reading a Datagram Socket

```
/* read_datagram.c    Use a datagram socket to read 5 messages from
                      write_datagram.c.  AF_UNIX domain: no networking.
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
    int sd,                                 /* Socket descriptor          */
        i,                                  /* for loop counter           */
        status;                             /* Status                     */
    char buf[80];                           /* Buffer to hold the messages */
    struct sockaddr_un sock_name;           /* Defined in <sys/un.h>      */

    strcpy(sock_name.sun_path, "/tmp/socket");   /* Declare the socket */
                        /* file by copying the name into sock_name */

    sd = socket(AF_UNIX,    /* Get a socket descriptor, AF_UNIX domain */
                SOCK_DGRAM,                                /* Datagram */
                0);                             /* Use default protocol */

    if ( sd == -1)
        perror("read_datagram.c: socket"), exit(1);

    status = bind(                  /* Bind the descriptor to the name */
                sd,                               /* Socket descriptor */
                &sock_name,         /* Structure containing the name */
                sizeof(sock_name));               /* Size of structure */

    if ( status == -1 )
        perror("read_datagram.c: bind"), exit(1);

    for (i = 0 ; i < 5 ; i++)       /* Read 5 messages from the socket */
    {
        status = recvfrom(                            /* Get a message */
                        sd,                       /* Socket descriptor */
                        buf,                              /* A message */
                        sizeof(buf),            /* Size of the message */
                        0, 0, 0);                          /* No flags */

        if ( status == -1 )
            perror("read_datagram.c: recvfrom"), exit(1);

        printf("%s\n", buf);                        /* Print a message */
    }

    close(sd);                                     /* Close the socket */

    unlink("/tmp/socket");        /* Remove the socket file (necessary */
                                  /* only in AF_UNIX domain)           */
}
```

The compiled version of `read_datagram.c` must be run first. Then `write_datagram.c` must be run on the same node.

Example 6-12: Writing to a Datagram Socket

```
/* write_datagram.c      datagram sockets employed to write 5 messages to
                         read_datagram.c.  AF_UNIX domain; no networking.
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
    int sd,                                   /* Socket descriptor       */
        i,                                    /* for loop counter        */
        status;                               /* sendto()'s status       */
    static char msg[] = "socket message ";    /* The basic message sent  */
    char str[20];                             /* The actual message sent */
    struct sockaddr_un sock_name;             /* Defined in <sys/un.h>   */

    strcpy(sock_name.sun_path,   /* Declare the socket file by copying */
           "/tmp/socket");       /* the name into sock_name.           */

    sd = socket(AF_UNIX,    /* Get a socket descriptor, AF_UNIX domain */
                SOCK_DGRAM,                              /* Datagram */
                0);                              /* Use default protocol */

    if ( sd == -1 )
        perror("write_datagram.c: socket"), exit(1);

    for (i = 0 ; i < 5 ; i++)/* Write 5 messages to socket in sendto() */
    {
        sprintf(str, "%s%d", msg, i);                 /* Form message */

        status = sendto(sd,                       /* Socket descriptor */
                        str,                                /* Message */
                        sizeof(str),                /* Size of message */
                        0,                                 /* No flags */
                        &sock_name, /* The sockaddr_un structure       */
                                    /* containing the socket file name */
                        sizeof(sock_name));       /* Size of structure */

        if ( status == -1 )
            perror("write_datagram.c: sendto"), exit(1);
    }

    close(sd);                                     /* Close the socket */

    unlink("/tmp/socket");        /* Remove the socket file (necessary */
                                  /* only in AF_UNIX domain)           */
}
```

6.3.2 Using a Stream Socket between Processes on the Same Node

The two example programs in this section communicate through a stream socket in the AF_UNIX domain (which means both processes must be on the same node). Both programs must obtain a socket descriptor and close the socket when they are through, but only one binds a name to the socket and removes the socket file after closing the socket, as shown in Table 6-3.

Table 6-3: System Calls for Stream Socket Communication

Server Program		Client Program
`sd = socket();`	Get a socket descriptor (like a file descriptor).	`sd = socket();`
`bind();`	Name the socket. The name is used by both processes to communicate through the same socket.	
`listen();`	Specify how many connection requests can be queued to this socket.	
`accept();`	Wait for a connection (or take the next one in the queue) and create a new socket descriptor for it. By default, the process blocks if no connections are pending, and unblocks when a connect request is made.	
	Request a socket connection.	`connect();`
`send(); recv();`	Send and receive messages. NOTE: not `sendto()` and `recvfrom()`, as with datagrams.	`send(); recv();`
`close();`	Close the socket.	`close();`
`unlink();`	Remove the socket file (AF_UNIX domain only). If the file is not removed, future attempts to bind to a socket of the same name will fail.	

The program in Example 6-13 creates an AF_UNIX domain stream socket and writes to it. The program in Example 6-14 reads everything from the socket in a single read; record boundaries are not preserved.

Example 6-13: Creating and Writing to a Stream Socket

```
/* write_stream.c       Create an AF_UNIX domain (same node) stream
                        socket and wait for read_stream.c to connect
                        to that socket.  Send 5 messages to read_stream.c.
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
    int sd,                                     /* Socket descriptor      */
    newsd,                                      /* New socket descriptor  */
    i,                                          /* for loop counter       */
    status;                                     /* Status                 */
    static char msg[] = "socket message";      /* The basic message sent  */
    char str[20];                               /* The actual message sent */
    struct sockaddr_un sock_name;               /* Defined in <sys/un.h>   */
```

Example 6-13: (continued)

```
    strcpy(sock_name.sun_path, "/tmp/stream");    /* Declare the socket */
                            /* file by copying the name into sock_name */

    sd = socket(AF_UNIX,     /* Get a socket descriptor, AF_UNIX domain */
                SOCK_STREAM,                             /* Stream socket */
                0);                                /* Use default protocol */

    if ( sd == -1 )
        perror("write_stream.c: socket"), exit(1);

    status = bind(                 /* Bind the descriptor to the name */
                  sd,                                 /* Socket descriptor */
                  &sock_name,          /* Structure containing the name */
                  sizeof(sock_name));                 /* Size of structure */

    if ( status == -1 )
        perror("write_stream.c: socket"), exit(1);

    if (listen(sd, 3) == -1) /* 3 connect requests can be queued to sd */
        perror("write_stream.c: listen"), exit(1);

    newsd = accept(sd,          /* Block, or accept the first pending */
                                /* connection, returning a new socket */
                                /* descriptor for the new connection. */
                   0,                   /* An unused result parameter */
                   0);                  /* An unused result parameter */

    if ( newsd == -1 )
        perror("write_stream.c: accept"), exit(1);

    for ( i = 0 ; i < 5 ; i++)               /* Write 5 messages to the */
                                             /* new socket descriptor   */
    {
        sprintf(str, "%s%d", msg, i);               /* Form the message */

        status = send(newsd,          /* Socket descriptor to write to */
                      str,                            /* String to write */
                      strlen(str),               /* Length of the string */
                      0);                                    /* No flags */

        if ( status == -1)
            perror("write_stream.c: send"), exit(1);
    }

    close(sd);                         /* Close the socket descriptor */
    close(newsd);                      /* Close the socket descriptor */
    unlink("/tmp/stream");                  /* Remove the socket file */
}
```

The compiled version of `write_stream.c` must be run first. Then `read_stream.c` must be run on the same node.

Example 6-14: Reading from a Stream Socket

```
/* read_stream.c      Read 5 messages in a single recv() call through
                      a stream socket.
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

Example 6-14: (continued)

```
main()
{
    int sd,                          /* Socket descriptor              */
    count,                           /* Number of bytes read by recv() */
    status;                          /* Status                         */
    char buf[200];                   /* A buffer to hold the message   */
    struct sockaddr_un sock_name;    /* Defined in <sys/un.h>          */

    strcpy(sock_name.sun_path, "/tmp/stream");   /* Declare the socket */
                            /* file by copying the name into sock_name */

    sd = socket(AF_UNIX,     /* Get a socket descriptor, AF_UNIX domain */
                SOCK_STREAM,                           /* Stream socket */
                0);                               /* Use default protocol */

    if ( sd == -1 )
        perror("read_stream.c: socket"), exit(1);

    status = connect(sd,          /* Block, or get a connection to sd */
                     &sock_name,      /* Structure containing the name */
                     sizeof(sock_name));           /* Size of structure */

    if (status == -1 )
        perror("read_stream.c: connect"), exit(1);

    sleep(3);            /* Allow writer time to send all the messages */

           /* Read all messages from server with one large recv() call */
    count = recv(  /* Returns number of bytes read (0 = closed socket) */
                 sd,                  /* Socket descriptor to read from */
                 buf,                    /* Buffer to hold what is read */
                 sizeof(buf),                     /* Size of the buffer */
                 0);                                         /* No flags */

    if ( count == -1 )
        perror("read_stream.c: recv"), exit(1);

    printf("%.*s\n\n", count, buf); /* Write buffer contents to stdout */

    close(sd);                           /* Close the socket descriptor */
}
```

6.3.3 Using a Stream Socket between Processes on Different Nodes

Sockets created in the AF_INET domain can communicate across a TCP/IP network. AF_INET sockets must have their names placed in a `sockaddr_in` structure, defined in `<netinet/in.h>`. Filling in the members of this structure is the chief difference between using an AF_INET domain socket and an AF_UNIX socket.

The program in Example 6-15 creates an AF_INET stream socket on node `rust`, then listens for connections. The program in Example 6-16 connects to that socket and writes to the process running the program in Example 6-15.

Example 6-15: Reading a Stream Socket Across an Internet Network

```
/* inet_server.c      Bind port 0x1234 on node "rust" and accept a
                      connection from inet_client.c on another node.
                      Read a message from inet_client.c and write it
                      to stdout.
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 0x1234       /* Pick a port number (like a descriptor), but */
                          /* 0x0-0x1023 is reserved for root             */
main()
{
    int sd,                             /* Socket descriptor             */
    newsd,                              /* Socket descriptor             */
    count;                              /* Number of bytes read by recv() */
    char buf[100];                      /* Buffer to hold the message read */
    struct sockaddr_in sin;             /* Defined in <netinet/in.h>      */

              /* Get a stream socket descriptor in the Internet domain */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        perror("inet_server.c: socket"), exit(1);

                                 /* Fill in the sockaddr_in structure */
    bzero(&sin, sizeof(sin));        /* Fill the structure with zeros */
    sin.sin_family = AF_INET;                      /* Internet domain */
    sin.sin_port = htons(PORT);  /* Convert host to network byte order */

    if (bind(sd, &sin, sizeof(sin)) == -1)      /* Bind the port number */
        perror("inet_server.c: bind"), exit(1);/* to the socket         */

                             /* Specify number of connection requests */
    if (listen(sd, 5) == -1)  /* that can be queued to server process  */
        perror("inet_server.c: listen"), exit(1);

    if ((newsd = accept(sd, 0, 0)) == -1)  /* Wait for connect request */
        perror("inet_server.c: accept"), exit(1);

                                  /* Read the message from the client */
    if ((count = recv(newsd, buf, sizeof(buf), 0)) == -1)
        perror("inet_server.c: recv"), exit(1);

    printf("Message received:\n%.*s\n", count, buf);  /* Print message */

    close(sd);                          /* Close the socket descriptor */
    close(newsd);                       /* Close the socket descriptor */
    exit(0);
}
```

The compiled version of `inet_server.c` must be run on node `rust` before `inet_client` is run. `inet_client` can be run on node `rust` or any node connected to `rust` by TCP/IP.

Example 6-16: Writing a Stream Socket Across an Internet Network

```
/*inet_client.c      Form a connection with inet_server.c at port 0x1234
                      on node "rust" and write a message to the server
*/

#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 0x1234      /* Pick a port number (like a descriptor), but */
                         /* 0-1023 is reserved for root                 */
main()
{
    int sd;                                        /* Socket descriptor */
    static char msg[] = "Socket message through the Internet domain";
    struct sockaddr_in sin;               /* Defined in <netinet/in.h> */
    struct hostent *hp; /* Defined in netdb.h; used by gethostbyname() */

             /* Put data about the remote node in a hostent structure */
    if ((hp = gethostbyname("rust")) == 0)
        perror("inet_client.c: gethostbyname"), exit(1);

                                  /* Fill in the sockaddr_in structure */
    bzero(&sin, sizeof(sin));          /* Fill the structure with zeros */
    bcopy(hp->h_addr, &sin.sin_addr, hp->h_length);/* Copy host address*/
    sin.sin_family = hp->h_addrtype;               /* Copy address type */
    sin.sin_port = htons(PORT);  /* Convert host to network byte order */

             /* Get a stream socket descriptor in the Internet domain */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        perror("inet_client.c: socket"), exit(1);

              /* Connect to the remote server (port 0x1234 at "rust") */
    if (connect(sd, &sin, sizeof(sin)) == -1)
        perror("inet_client.c: connect"), exit(1);

    if (send(sd, msg, sizeof(msg), 0) == -1)     /* Write a message to */
        perror("inet_client.c: write"), exit(1); /* the remote server  */

    close(sd);
    exit(0);
}
```

6.3.4 Socket Flow Control

Processes communicating through sockets can be blocked (prevented from executing further). A socket-writing process is blocked if more than 4K bytes remain unread in the socket. A socket-reading process is blocked if the socket is empty. To avoid being blocked under these conditions, programs can use the `fcntl()` system call. To determine if a socket is ready for I/O, and to optionally be blocked if it is not, programs can use the `select()` system call.

A process that writes to a socket with no readers receives a SIGPIPE signal. Without a SIGPIPE signal handler, the process terminates.

6.3.4.1 Using fcntl( ) to Prevent Blocking

During I/O, the `fcntl()` system call can prevent a process from being blocked to a single descriptor.

```
#include <fcntl.h>
     .
     .
     .
result = fcntl (sd,         /* Socket to act on                        */
                F_SETFL,    /* Set flags (the flag is FNDELAY)          */
                FNDELAY);   /* No delay.  If read or write call would */
                            /* block, the read or write call returns  */
                            /* -1 and sets errno to EWOULDBLOCK.  The */
                            /* program continues                      */
```

After executing the previous code to prevent being blocked, the process can continually try the I/O until it is possible. However, using the FASYNC flag causes a SIGIO signal to be sent to the process when non-blocking I/O is possible:

```
#include <fcntl.h>
     .
     .
     .
result = fcntl (sd,        /* Socket to act on                    */
                F_SETFL,   /* Set flags (the flag is FASYNC)      */
                FASYNC);   /* Send a SIGIO signal to the process */
                           /* when non-blocking I/O is possible  */
```

A program that uses the preceding call to `fcntl()` must have a SIGIO signal handler.

6.3.4.2 Using select( ) to Determine Descriptor Status

The `select()` system call determines whether any of a set of descriptors is ready for I/O; for example:

```
#include <sys/time.h>

struct timeval *timeout;
int nfound, num_fds, *readfds, *writefds, *execptfds;

nfound = select(            /* Returns the number of ready descriptors */
                num_fds,    /* Check first num_fds bits (descriptors)  */
                readfds,    /* Bit mask of descriptors to be read      */
                writefds,   /* Bit mask of descriptors to be written   */
                execptfds,  /* Bit mask of descriptors with exception  */
                            /*   conditions pending                    */
                timeout);   /* How much time to wait for select()      */
```

Descriptor *n* is checked if bit *n* is 1 in any of the three bitmasks. Therefore, descriptor `sd` can be added to a bit mask using the following statement:

```
mask |= 1 << sd;
```

The `select()` call clears bits from masks for the descriptors that are not ready; that is, the bit mask parameters receive return arguments that are bit masks depicting which descriptors are ready for I/O.

If `timeout` is 0, `select()` blocks the process until a descriptor specified in one of the mask arguments is ready for I/O. Otherwise, `timeout` must be a pointer to a timeval structure (defined in <sys/time.h>):

```
struct timeval {
        long    tv_sec;         /* Seconds */
        long    tv_usec;        /* Microseconds */
};
```

If the values in the timeval structure are 0, select() polls each descriptor only once. Otherwise, the descriptors are polled continuously until the specified time has elapsed.

The program in Example 6-17 makes an AF_UNIX stream socket, accepts a connection, waits awhile, then writes to the socket. The program in Example 6-18 connects to the socket and uses select() to block the process until something is written to the socket.

Example 6-17: A Slow Socket Writer

```
/* slow_writer.c       Make an AF_UNIX stream socket, accept a connection,
                       waste some time, then send a message.
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
    int sd, newsd;                              /* Socket descriptors */
    static char message[] = "Was this message worth waiting for?";
    struct sockaddr_un sock_name;             /* Defined in <sys/un.h> */

    strcpy(sock_name.sun_path, "/tmp/socket");      /* Name the socket */

                                          /* Get a socket descriptor */
    if ( (sd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 )
        perror("slow_writer.c: socket"), exit(1);

                                  /* Bind the descriptor to the name */
    if ( bind(sd, &sock_name, sizeof(sock_name)) == -1 )
        perror("slow_writer.c: bind"), exit(1);

          /* Make the descriptor's connection queue 5 connections long */
    if ( listen(sd, 5) == -1 )
        perror("slow_writer.c: listen"), exit(1);
                             /* Block until there is a connection request */
    if ( (newsd = accept(sd, 0, 0)) == -1 )
        perror("slow_writer.c: accept"), exit(1);

    sleep(5);                                       /* Waste 5 seconds */

    if ( send(newsd, message, strlen(message), 0)    /* Send a message */
         == -1 )
        perror("slow_writer.c: send"), exit(1);

    close(sd);                                 /* Close the descriptor */
    close(newsd);                              /* Close the descriptor */
    unlink("/tmp/socket");                   /* Remove the socket file */
}
```

Example 6-18: Using select( ) to Wait on a Stream Socket

```
/* patient_reader.c      Use select() to determine when a message can be
                         read from an AF_UNIX domain stream socket
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

main()
{
    int  sd,              /* Socket descriptor                          */
         count,           /* Number of bytes returned by recv()         */
         nfound,          /* Number of descriptors found by select()    */
         rmask = 0;       /* Bit mask of descriptors ready for reading */
    char buf[80];         /* Buffer to hold the message read            */
    struct sockaddr_un sock_name;             /* Defined in <sys/un.h> */

    strcpy(sock_name.sun_path, "/tmp/socket");      /* Name the socket */

                                             /* Get a socket descriptor */
    if ( (sd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 )
        perror("patient_reader.c: socket"), exit(1);

                                                /* Request a connection */
    if ( connect(sd, &sock_name, sizeof(sock_name)) == -1 )
        perror("patient_reader.c: connect"), exit(1);

    rmask |= 1 << sd;             /* Set the bit for the sd descriptor */

    nfound = select(       /* Returns the number of ready descriptors  */
                    32,    /* Check descriptors (bits) 0-31             */
                    &rmask,/* Descriptors checked for read readiness    */
                    0,     /* Check no descriptors for write readiness */
                    0,     /* Check no descriptors for exceptions       */
                    0);    /* Block until sd can be read                */

    if ( nfound == -1 )
        perror("patient_reader.c: select"), exit(1);

        /* After select unblocks, print some data and read the message */
    printf("Number of desc. found = %d\nReturned select mask = %x\n",
        nfound, rmask);

    if ( (count = recv(sd, buf, sizeof(buf), 0)) == -1 )
        perror("patient_reader.c:  recv"), exit(1);

                                                   /* Print the message */
    printf("Message through socket:  %.*s\n", count, buf );

    close(sd);                                  /* Close the descriptor */
}
```

The `slow_writer` program (Example 6-17) must be run before the `patient_reader` program (Example 6-18). Both programs must be run on the same node.

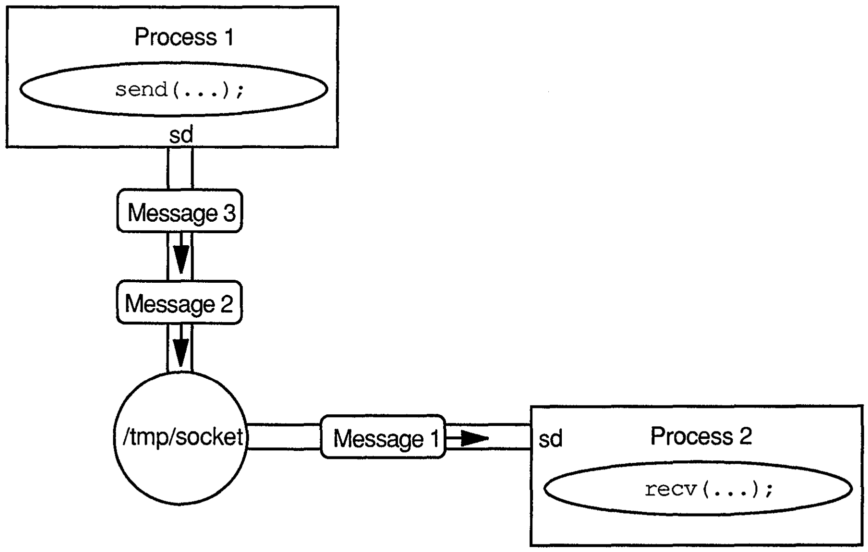
6.3.5 Special Stream Socket Features

Stream socket messages have two special abilities: the ability to be peeked at, and the ability to skip to the front of the message queue (out-of-bounds messages).

6.3.5.1 Peeking at a Message

Figure 6-9 shows a typical message queue between two processes communicating through an AF_UNIX domain stream socket.

Figure 6-9: Socket Message Queue



The reading process receives the data in the order sent. When a message is read, it is removed from the queue. But when a message is peeked at, it remains in the queue. The peeking process gets a preview of the next message without reading it and, hence, removing it from the queue. Peeking is accomplished with the `MSG_PEEK` argument to `recv()`:

```
count = recv(sd,           /* Socket descriptor to peek                 */
             buf,          /* Character array to hold the message       */
             sizeof(buf),  /* How many bytes to read                    */
             MSG_PEEK);    /* Just peek; don't remove bytes from socket */
```

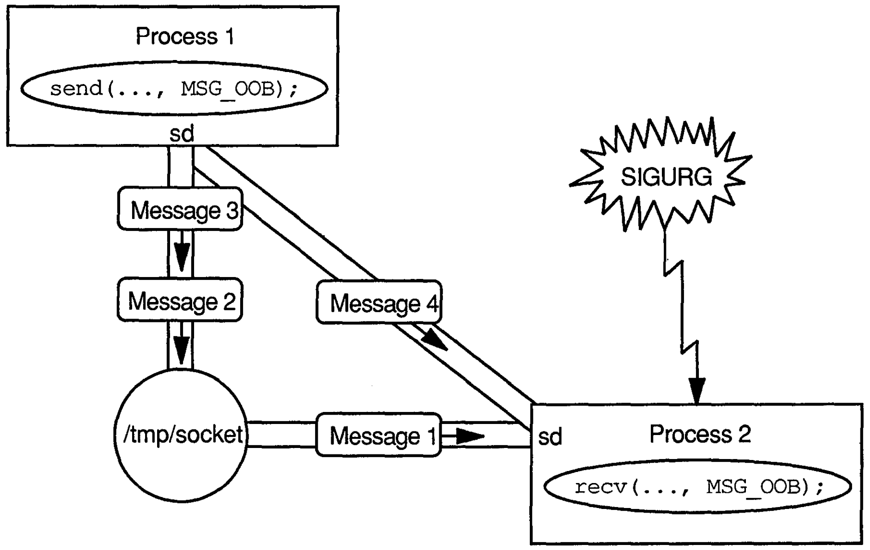
6.3.5.2 Sending and Receiving Out-of-Bounds Messages

A process writing to a socket can send an out-of-bounds message to that socket by using the `MSG_OOB` argument to `send()`:

```
status = send(newsd,           /* Socket descriptor to write to  */
             message,          /* Message to write               */
             sizeof(message),  /* How many bytes to write        */
             MSG_OOB);         /* Send the message out of bounds */
```

An out-of-bounds message uses a parallel socket channel, bypassing all messages in the queue. In Figure 6-10, process 1 writes three normal messages to a socket, and then writes an out-of-bounds message (message 4) to that socket. Process 2, which has connected to that socket, receives a SIGURG signal, meaning that an out-of-bounds message has been sent to the socket it is connected to. Process 2 can now read either message 1, or the out-of-bounds message 4. Typically, a SIGURG signal handler is invoked by a reading process to read out-of-bounds messages.

Figure 6-10: Out-of-Bounds Message



The process reading the stream socket receives a SIGURG signal when an out-of-bounds message arrives. Out-of-bounds messages are read by using the `MSG_OOB` argument to `recv()`:

```
count = recv(sd,             /* Socket descriptor to peek         */
             buf,            /* Character array to hold the message */
             sizeof(buf),    /* How many bytes to read            */
             MSG_OOB);       /* Read the out-of-bounds message,   */
                             /*   not the next regular message    */
```

6.3.6 Additional Socket Information

The following sections discuss ancillary socket features.

6.3.6.1 Special Socket Options

The `getsockopt()` and `setsockopt()` system calls can be used to get and set special socket options. For more information, see `setsockopt`(2) and `getsockopt`(2) in the *ULTRIX Reference Pages.*

6.3.6.2 Using read() and write()

The `read()` and `write()` system calls can be used on socket descriptors just as they are on file descriptors. However, out-of-band messages and message peeking cannot be used.

6.3.6.3 Closing Halves of a Socket

Sockets are full duplex connections (read and write both directions). To close half of a socket, making it read-only or write-only for a particular process, the `shutdown()` system call can be used; for example:

```
shutdown(sd,  /* The socket descriptor to act on               */
         0);  /* 0 stops reading, 1 stops writing, 2 stops both */
```

Any pending `send()`, `recv()`, `write()`, and `read()` calls are flushed (not discarded), so messages are not lost.

Writing Secure Programs 7

This chapter presents security guidelines for these programming tasks:

- Using open file descriptors
- Responding to signals
- Specifying a secure search path
- Protecting permanent and temporary files
- Handling errors
- Using privileged processes
- Writing SUID and SGID programs
- Authenticating users
- Writing shell scripts and protecting compiled programs
- Programming in a DECwindows environment

This chapter also discusses the ULTRIX system calls and library routines that have security implications. All system calls and library routines discussed in this chapter are documented in the *ULTRIX Reference Pages.*

7.1 Using Open File Descriptors with Child Processes

A child process can inherit all the open file descriptors of its parent process and therefore can have the same type of access to files that the parent has. This relationship creates a security concern.

For example, suppose you write a set user ID (SUID) program that does the following:

- Allows users to write data to a sensitive, privileged file
- Creates a child process that runs in a nonprivileged state

Because the parent SUID process opens a file for writing, the child (or any user running the child process) can write to that sensitive file.

To protect sensitive, privileged files from users of a child process, close all file descriptors that are not needed by the child process before the child is created. An efficient way to close file descriptors before creating a child process is to use the `fcntl` system call. You can use this call to set the `close-on-exec` flag on the file after you open it. File descriptors that have this flag set are automatically closed when the process `exec's` a new program.

For more information, see `fcntl`(2).

7.2 Responding to Signals

The ULTRIX operating system generates signals in response to certain events. The event could be initiated by a user at a terminal (such as quit, interrupt, or stop), by a program error (such as a bus error), or by another program (such as kill).

By default, most signals terminate the receiving process; however, some signals only stop the receiving process. Many signals, such as SIGQUIT or SIGTRAP, write the core image to a file for debugging purposes. A core image file might contain sensitive information, such as passwords.

To protect sensitive information in core image files and protect programs from being interrupted by input from the keyboard, write programs that capture signals such as SIGQUIT, SIGTRAP, or SIGTSTP. Use the `signal` routine to cause your process to change its response to a signal. This routine enables a process to ignore a signal or call a subroutine when the signal is delivered. (The SIGKILL and SIGSTOP signals cannot be caught, ignored, or blocked. They are always passed to the receiving process.) For more information, see `signal`(3) and `sigvec`(2).

Also be aware that child processes inherit the signal mask that the parent process sets before calling `fork`. The `execve` system call resets all caught signals to the default action; ignored signals remain ignored. Therefore, be sure that processes handle signals appropriately before you call `fork` or `execve`. For more information, see `fork`(2) and `execve`(2).

7.3 Specifying a Secure Search Path

If you use the `popen`, `system`, or `exec*p` routines, which execute `/bin/sh`, be careful when specifying a pathname or defining the shell `PATH` variable. The `PATH` variable is a security-sensitive variable because it specifies the search path for executing commands and scripts on your system. For more information, see `environ`(7), `popen`(3), and `system`(3).

The following list describes how to create a secure search path:

- Specify absolute path names for the `PATH` variable.
- Do not include public or temporary directories, other users' directories, or the current working directory in your search path. Including these directories increases the possibility of inadvertently executing the wrong program or of being trapped by a malicious program.
- Be sure that system directories appear before user directories in the list. This prevents you from mistakenly executing a program that might have the same name as a system program.
- Analyze your path list syntax, especially use of nulls, decimal points, and colons. A null entry or decimal point entry in a path list specifies the current working directory and a colon is used to separate entries in the path list. For this reason, the first entry following an equal sign should never begin with a colon.
- If there is a colon at the end of the path list, certain shells and `exec*p` search the current working directory last. To avoid having various shells interpret this trailing colon in different ways, use the decimal point to reference the current working directory rather than using a null entry to reference the current working directory.

You might want to use the `execve` system call rather than any of the `exec*p` routines because `execve` requires that you specify the pathname. For more information, see `execve`(2).

7.4 Protecting Permanent and Temporary Files

If your program uses any permanent files (for example, a database), make sure that these files have restrictive permissions and that your program provides controlled access. These precautions also apply to shared memory segments, semaphores, and interprocess communication mechanisms; set restrictive permissions on all of these objects.

Programs sometimes create temporary files to store data while the program is running. Follow these precautions when you use temporary files:

- Be sure your program deletes temporary files before it exits.
- Avoid storing sensitive information in temporary files, unless the information has been encrypted.
- Give only the owner of the temporary file read and write permission. Set the file creation mask to 077 by using the `umask`(2) system call at the beginning of the program.
- Create temporary files in private directories that are writable only by the owner. If you must use `/tmp`, ask your security administrator to set the sticky bit on the directory (mode 1777), so that files in it can be deleted only by the file owner, the owner of the directory, or the superuser.

A common practice is to create a temporary file, then unlink the file while it is still open. This limits access to any processes that had the file open before the unlink; when the processes exit, the inode is released.

Note that this use of unlink on an NFS-mounted file system takes a slightly different action. The client kernel renames the file and the unlink is sent to NFS only when the process exits. You cannot guarantee that the file will be inaccessible to someone else, but you can be reasonably sure that the file will be inaccessible when the process exits. In any case, always explicitly ensure that no temporary files remain after the process exits.

7.5 Handling Errors

Most system calls and library routines return an integer return code, which indicates the success or failure of the call. Always check the return code to make sure that a routine succeeded. If the call fails, test the global variable `errno` to find out why it failed.

The `errno` variable is set when an error occurs in a system call. You can use this value to obtain a more detailed description of the error condition. This information can help the program decide how to respond, or produce a more helpful diagnostic message. This error code corresponds to an error name in `<errno.h>`. For more information, see `errno`(2).

The following `errno` values indicate a possible security breach:

EPERM	Indicates an attempt by someone other than the owner to modify a file in a way reserved to the file owner or superuser. It can also mean that a user attempted to do something that is reserved for a superuser.

EACCES	Indicates an attempt to access a file for which the user does not have permission.
EROFS	Indicates an attempt to access a file on a mounted file system when that permission has been revoked.

If your program makes a privileged system call, but the resulting executable program does not have superuser privilege, it will fail when it tries to execute the privileged system call. If the security administrator has set up the audit system to log failed attempts to execute privileged system calls, the failure will be audited.

If your program detects a possible security breach, do not have it display a diagnostic message that could help an attacker defeat the program. For instance, do not display a message that indicates the program is about to exit because the attacker's real user ID (UID) did not match a UID in an access file, or even worse, go on to provide the name of the access file. In addition, you could program a small delay before issuing a message to prevent programmed attempts to penetrate your program by systematically trying various inputs.

7.6 Using Privileged Processes

Any process that runs with an effective UID of 0 is a privileged process. A process runs with an effective UID of 0 if one of the following is true:

- The process executing the program is a superuser process
- The program's UID is set to `root` and the SETUID bit is set

You must be alert to the fact that some system calls and library routines, when called by a privileged process, behave differently than the way they behave when called by a nonprivileged process.

For example, the `setuid` routine sets both the real and effective UIDs, and the `setgid` routine sets both the real and effective group IDs (GIDs). A nonprivileged process can only set the effective UID to the real UID. A privileged process is not restricted in this fashion and can set these values as it chooses. For more information, see `setuid`(3) and `setgid`(3).

Additionally, some system calls can only be called from a privileged process. For example, only a privileged process can call `sethostid` or `chroot`. For more information, see `sethostid`(2) and `chroot`(2).

All system calls bypass file protections when called from a privileged process. The following list provides some examples of system calls that behave differently from their nonprivileged behavior when called from a privileged process, or can be called only by a privileged process:

Restricted to root	`acct`, `adjtime`, `audcntl`, `audgen`, `chroot`, `exportfs`, `setdomainname`, `sethostid`, `sethostname`, `settimeofday`, `plock`, `reboot`, `setgroups`, `setquota`, `stime`, `swapon`, and `vhangup`.
Different for root	`bind`, `chown`, `setpriority`, `setrlimit`, `kill`, `killpg`, `link`, `mknod`, `mount`, `quota`, `setpgrp`, `setregid`, `setreuid`, `setsysinfo`, `socket`, and `ulimit`.

Bypass permissions	`msgctl`, `msgsnd`, `msgrcv`, `semctl`, `semop`, `shmctl`, `shmat`, and any calls that use file-system pathnames.

Make sure that your compile environment (BSD, SYSTEM_FIVE, POSIX, or X_OPEN) does not change the behavior that you expect from a system call or library routine.

7.6.1 Use Minimum Privileges

Because a privileged process has extraordinary powers, create a program that runs as a privileged process only if there is no other way to accomplish the task, and remove superuser privileges (the program's UID is not `root`) when the process no longer requires them.

Once a privileged process uses the `setreuid` system call to change its real and effective UIDs to something other than 0, it cannot regain superuser privileges. If you write a program that performs both privileged and nonprivileged operations and plan to use `setreuid` to reduce the amount of time the process spends in a privileged state, remember to perform all privileged operations before calling `setreuid.` For more information, see `setreuid`(2).

Another approach is to have the program retain superuser privileges and create child processes for nonprivileged operations. Each child process would call `setreuid` to give up its privileged status. This separates privileged from nonprivileged operations within the program, reducing the potential for error or compromise while in a privileged state.

7.6.2 Use Care When Allocating System Resources

Privileged programs can deliberately or accidentally have a negative effect on the services available to users. For example, privileged programs can call `ulimit` and `nice` to increase file-size limits and set higher priorities for themselves. These changes might have the side effect of denying services to users. Therefore, be careful when you allocate system resources or change system parameters; check for side effects to avoid monopolizing system resources. For more information, see `ulimit`(2) and `nice`(1).

7.6.3 Know the Process's Real UID

Before performing certain privileged operations, you might want to know who is actually running the program. Use the `getuid` system call to determine the real UID associated with the process. To decide whether or not to allow access to a file, use the `access` system call to determine if the real UID (the user) could access the file in question without the power of a privileged process. You can use this call to decide when to limit the inherent access privileges associated with an effective UID of 0. For more information, see `getuid`(2) and `access`(2).

7.6.4 Audit Security-Relevant Events

If your security administrator has enabled security auditing, the audit daemon, `auditd`, reads data from `/dev/audit` and stores that data in the auditlog. The audit subsystem can record a wide range of system events. The security administrator can choose events to be logged. For more information, see `auditd`(8) and `auditmask`(8). For a complete description of the audit subsystem, see the *Security Guide for Administrators*.

You might want to write a program that generates an audit record for process events. You might also want to change the events that are audited or the items that are recorded in an audit record for a process. The following privileged system calls and library routine enable you to interact with the audit subsystem:

audgen	This system call generates an audit record for a specified operation or event and stores it in the auditlog. Your process can call `audgen` directly, or it can use the library routine, `audgenl`, for this operation. For more information, see `audgen`(2) and `audgenl`(3x).
audcntl	Provides control over options offered by the audit subsystem. For more information, see `audcntl`(2).

Audit record generation depends on a combination of the system audit mask and the process audit mask. Each process has an audit mask and an audit control flag, both of which originate in `/etc/auth`. Each event that can be audited is represented in both the system and the process masks. Whether the event is audited depends on the audit control flag, as described in the following list:

- If the process audit control flag is set to `AND`, both masks must indicate that the event is to be audited.
- If the process audit control flag is set to `OR`, at least one of the masks must indicate that the event is to be audited.
- If the process audit control flag is set to `OFF`, no events for the process are audited.
- If the process audit control flag is set to `USR`, the process is audited according to the process mask only.

The following example shows how a privileged program turns off auditing for the current process only:

```
/* Turns off auditing for this process */
# include <sys/audit.h>
audcntl (SET_PROC_ACNTL, (char *)0, 0, AUDIT_OFF, 0);
```

Example 7-1 shows two ways a privileged program can generate an audit record: by using the `audgen` system call, and by using the `audgenl` library routine.

Example 7-1: Two Ways to Generate an Audit Record

```
/* audgen system call to generate a sample audit record */

#include <sys/audit.h>

main()
{
    char tmask[AUD_NPARAM];
    struct {
        char *a;
        int b;
    } aud_arg;
    int i;

    tmask[0] = T_CHARP;
    tmask[1] = T_ERROR;
    tmask[2] = ' ';
```

Example 7-1: (continued)

```
    aud_arg.a = "anything you want to put in the record";
    aud_arg.b = -1;

    if ( audgen ( AUTH_EVENT, tmask, &aud_arg ) == -1 )
        perror ( "audgen" );
}


/* audgenl library routine to generate the same sample audit record  */

#include <sys/audit.h>

main()
{
    if ( audgenl ( AUTH_EVENT, T_CHARP, "any string", T_CHARP,
    "anything you want to put in the record", T_ERROR, -1, 0 ) == -1 )
        perror ( "audgenl" );
}
```

In Example 7-1, the `argv` argument is a pointer to an argument vector. Each entry in the token type array describes the corresponding entry in the argument vector. In this example, `T_CHARP` is a token type describing the string "anything you want to put in the record". `T_ERROR` is a token type associated with the error value of –1. You can create an audit record containing up to eight token types and values.

In Example 7-2, a privileged program uses the `audcntl` system call to change the events that are audited for this process. This example shows how to adjust the process audit control flag.

Example 7-2: Using the audcntl Call to Change the Audit Control Flag

```
/* Change the events that are audited for this process */
# include <syscall.h>
# include <sys/audit.h>
# define LEN (SYSCALL_MASK_LEN+TRUSTED_MASK_LEN)
    char buf[LEN];
/* Change process mask to specify auditing for login and failed
 * setgroups (note that 'buf' is initially set to zero). The
 * process mask is AND'ed with the system mask. This results
 * in only LOGIN and SYS_setgroups being audited for this process
 * (and only if the system mask also specifies LOGIN and/or
 * SYS_setgroups).
 */

/* Get process control flag to AND */
if (audcntl (SET_PROC_ACNTL, (char *)0, 0, AUDIT_AND, 0) == -1)
     perror ("audcntl");

/* Get process mask */
A_PROCMASK_SET (buf, SYS_setgroups, 0, 1 );
A_PROCMASK_SET (buf, LOGIN, 1, 1 );
if (audcntl (SET_PROC_AMASK, buf, LEN, 0, 0 ) == -1 )
     perror ("audcntl");
```

In Example 7-2, the `A_PROCMASK_SET` macro, from `audit.h`, takes the following arguments:

1. The buffer into which the mask is being built (`buf`).
2. The event name, from `syscall.h` for system calls and `audit.h` for events (SYS_setgroups and LOGIN)
3. An integer (1=yes, 0=no) that indicates whether a successful occurrence of the event should be audited.

4. An integer (1=yes, 0=no) that indicates whether a failed occurrence of the event should be audited.

7.6.5 Use Care When Creating Daemons as Privileged Programs

Daemons are long-lived, background processes that provide system-related services. Some standard daemons are the `swapper`, `pagedaemon`, `cron`, `elcsd`, and `lpd` daemons. Daemons do not necessarily have to be privileged programs; however, most daemons require privileged access to carry out their tasks. If you create a daemon as a privileged program, take the same care as with any other privileged program. The following list describes some ways to make your privileged daemons more secure:

- Check who is actually requesting the service. Note that this can be a problem if the connection is through a socket, because the information about who is requesting the service is not available from a socket.

  The best approach for safely using sockets in privileged daemons is as follows:

  – Use INET-domain sockets. If you must use UNIX sockets, place the sockets in a protected directory.

  – Have the daemon check that the other side of the connection is a privileged port. A socket can be marked privileged only if the superuser created it. Only privileged sockets can send broadcast packets or bind addresses in privileged portions of an address space. The daemon can determine whether the other side of the connection is a privileged port through the `accept` or `getpeername` system calls. For more information, see `accept`(2) and `getpeername`(2).

  – Write an auxiliary privileged program that connects to the daemon using a privileged port. For example, the auxiliary program can use the `rresvport` routine to get a privileged port. This requires superuser access. For more information, see `rresvport`(3). This auxiliary program can perform checks on the user, because it knows who invoked it (either from the audit UID or the real UID). The auxiliary program can then communicate this information to the daemon. The daemon refuses to accept any connection that is not from the auxiliary program.

- Remove the controlling terminal using the `ioctl(fd,TIOCNOTTY)` function call.

- Create separate processes for nonprivileged tasks, and remove privileges at the beginning of the routines. If you have separate programs that work with the daemon, in the same way that `lpr` works with `lpd`, make sure that the interaction between the programs cannot be exploited to create a security breach. Put proper protections on both programs. For more information, see `lpr`(1).

- Put proper ownership and protections on any permanent or temporary files. Clean up any temporary files before exiting. You might want to use a directory other than `/tmp`, depending on the number of files and security issues. Make sure that only the daemon can write to any important directories (or that the sticky bit is set). You might want to create a separate account for the daemon in order to control file ownership and access.

7.7 Writing SUID and SGID Programs

SUID and SGID programs change the effective UID or GID of a process to the UID or GID of the program. They are a solution to the problem of providing controlled access to system-level files and directories, because they grant a process the access rights of the files' owner.

The potential for security abuse is higher for programs in which the user ID is set to `root` or the group ID is set to any group that provides write access to system-level files. Simply stated, do not write a program that sets the user ID to `root` unless there is no other way to accomplish the task. If you must write a program that sets the user ID to `root`, refer to Section 7.6 for information about writing secure privileged programs.

The `chown` system call automatically removes any SUID or SGID bits on a file, unless the RUID of the executing process is set to zero. This prevents the accidental creation of SUID or SGID programs owned by the `root` account. For more information, see `chown`(2).

The following list provides suggestions for creating more secure SUID and SGID programs:

- Verify all user-provided pathnames with the `access` system call.
- Trap all relevant signals to prevent core dumps.
- Test for all error conditions, such as system call return values and buffer overflow.

When possible, create SGID programs rather than SUID programs. One reason is that file access is generally more restrictive for a group than for a user. If your SGID program is compromised, the restrictive file access reduces the range of actions available to the attacker.

Another reason is that it is easier to access files owned by the user executing the SGID program. When a user executes an SUID program, the original effective UID is no longer available for use for file access. However, when a user executes an SGID program, the user's primary GID is still available as part of the group access list. Therefore, the SGID process still has group access to the files that the primary GID could access.

7.8 Authenticating Users

You need access to the following to authenticate a user on an ULTRIX system:

- Username
- Password
- `/etc/passwd` file
- `/etc/svc.conf` file

The system administrator may optionally configure the system to store the passwords for each account in a database, `auth`, which is not accessible to unprivileged processes. In addition to the password, this database contains much additional information about the user, including password expiration information. The contents of the file `/etc/svc.conf` determines if this database is to be used.

There are two ways to authenticate a user on an ULTRIX system. The simple way is to use the `authenticate_user` library routine, as documented in `authenticate_user`(3x).

Another method to authenticate a user is to follow these steps (although using `authenticate_user` performs the same checks and produces the same results):

1. Use the `getpwnam` library routine to get the `passwd` database entry corresponding to the username. For information, see `getpwnam`(3).
2. Use the `getsvc` library routine to get security information from the `/etc/svc.conf` file. For information, see `getsvc`(3).
3. Check the value of the `seclevel` field in the `/etc/svc.conf` file to determine where the password is stored and whether password expiration information is available.
   - If the security level is `SEC_BSD`, the password is stored in the `passwd` database. No password expiration information is available.
   - If the security level is `SEC_UPGRADE`, password expiration information is available. The password is usually stored in the `passwd` database. The exception is if the entry in the `passwd` database is exactly equal to the string "*". In this case, the password is stored in the `auth` database.
   - If the security level is `SEC_ENHANCED`, password expiration information is available and the password is always stored in the `auth` database.
4. Encrypt the password supplied using the first two characters of the old encrypted password as the `salt` argument.
   - If the password came from the `passwd` database, use the `crypt` library routine. For information about encrypting passwords and using `salt`, see `crypt`(3.)
   - If the password came from the `auth` database, use the `crypt16` function. For information about encrypting passwords with the `crypt16` function and using `salt`, see `crypt`(3).
5. Compare the encrypted password the user entered with the password in the `passwd` or `auth` database. If the two encrypted passwords match, the password is valid.
6. If password expiration information is available, further verify the password by testing that the password has not expired.

   To perform this test, check the password modification time stored in the `auth` database record against the maximum password lifetime information, which is also stored there, using the current system time as a reference. If modification time plus maximum lifetime is less than the current system time, the password has expired and the account is not valid.

   An additional time factor, called the soft expiration time, can also be used in the calculation to provide a grace period during which users can log into the system provided they change their passwords immediately.
7. Depending on your application, you may also want to check the `A_LOGIN` flag in the `auth` database record for the user.

Example 7-3 shows a routine that authenticates a user's password.

Example 7-3: Routine to Authenticate a User

```
/*
 * authenticate - a routine to verify a user's password.
 */
#include <pwd.h>
#include <sys/svcinfo.h>
#include <auth.h>
int authenticate(username, passwd)
char *username, *passwd;
{
        struct passwd *pwd, *getpwnam();
        AUTHORIZATION *auth, *getauthuid();
        char *pp, *crypt(), *crypt16(), *(*fp)();
        struct svcinfo *svcinfo;
        auth = (AUTHORIZATION *) 0;
        if(!(pwd=getpwnam(username)))
                return 0; /* no account */
        if(!(svcinfo=getsvc()))
                return 0; /* should never happen */
        switch(svcinfo->svcauth.seclevel) {
        case SEC_BSD:
                pp = pwd->pw_passwd;
                fp = crypt;
                break;
        case SEC_UPGRADE:
                if(!(auth=getauthuid(pwd->pw_uid)))
                        return 0; /* no auth entry */
                if(!strcmp(pwd->pw_passwd, "*")) {
                        pp = auth->a_password;
                        fp = crypt16;
                } else {
                        pp = pwd->pw_passwd;
                        fp = crypt;
                }
                break;
        case SEC_ENHANCED:
                if(!(auth=getauthuid(pwd->pw_uid)))
                        return 0; /* no auth entry */
                pp = auth->a_password;
                fp = crypt16;
                break;
        default:
                return 0; /* bad seclevel in /etc/svc.conf */
        }
        if(!*pp && *passwd)
                return 0; /* bad password */
        if(strcmp((*fp)(passwd, pp), pp))
                return 0; /* bad password */
        if(auth) {
                long expiration, time();
                if(auth->a_pw_maxexp) {
                        expiration = auth->a_pass_mod + auth->a_pw_maxexp;
                        if(time((long *) 0) > expiration)
                             return 0; /* password expired */
                }
                if(!(auth->a_authmask & A_LOGIN))
                        return 0; /* account disabled */
        }
        return 1;
}
```

Note

Although the `authenticate_user`, `getpwnam`, and `getauthuid` library functions transparently retrieve entries served from remote hosts, you must get a Kerberos ticket-granting `ticket` before you can obtain `auth` database entries for hosts served through BIND/Hesiod. See the *Guide to Network Programming* for information about using Kerberos.

7.9 Protecting Shell Scripts and Compiled Programs

A shell script is a file containing shell commands. Shell scripts can include variables and flow control constructions. If you must use a shell script to handle sensitive data, set and export `path` before writing the body of the script. Do not make shell scripts SUID or SGID.

Compiled programs enjoy a measure of security that shell scripts do not. You can allow users to execute compiled programs while restricting those users from reading the source files. Because users need both read and execute permission to run a shell script, they have a much better chance of deciphering and compromising the script. For this reason, compile any program whose compromise represents a security risk and make it available to the general user only as an executable file.

Deny access to any source files. Remove read permission for group and other on the executable file to deny users the opportunity to use a debugger on the file.

7.10 Security Concerns when Programming in a DECwindows Environment

The following sections discuss four ways to increase security in a DECwindows programming environment:

- Restrict access control
- Protect keyboard input
- Block keyboard and mouse events
- Protect device-related events

For a detailed description of `Xlib` library calls and the X Window System Protocol, see the *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD.*

7.10.1 Restrict Access Control

The access control list is the key to security in the DECwindows environment. This list names the hosts on the network that can access a workstation display. Users logged in to hosts listed in the access control list can read from, write to, and copy the contents of any window by specifying the window ID. Unlike files, windows cannot be protected from authorized users by setting permissions on them.

When a system is installed, the only host listed in the access control list is the local host. The local host is the system on which the window system is running. For example, when workstation `rook` is booted for the first time, `rook` is the only host listed in its access control list.

The system access control list is stored in a privileged file called `/etc/X*.hosts`. The asterisk specifies the number of the workstation display. When a system is installed, this file is either empty or does not exist. The security administrator maintains this file, usually by leaving it empty to protect the workstations on the network from security attacks. If a user does not add any hosts to the workstation access control list (using the Security option from the Customize menu) the `/etc/X*.Hosts` file determines the access control list for that workstation.

Table 7-1 lists the `Xlib` library function calls that maintain the access control list for a local worksystem display. That is, the function calls in the table can be executed only for the host where the access list is to be changed.

Table 7-1: Xlib Library Function Calls That Maintain the Access Control List of A Local Worksystem Display

Call	Purpose
`XAddHost`	Add a single host to the access control list for the workstation display.
`XAddHosts`	Add the specified hosts to the access control list for the workstation display.
`XListHosts`	List the hosts on the access control list of the workstation display. This call enables a program to find out which hosts can connect to the workstation display.
`XRemoveHost`	Remove the specified host from the access control list for the workstation display.
`XRemoveHosts`	Remove the specified hosts from the access control list for the workstation display.
`XEnableAccessControl`	Enable the use of the access control list at the workstation.
`XDisableAccessControl`	Disable the use of the access control list at the workstation.

7.10.2 Protect Keyboard Input

Users logged into hosts listed in the access control list can call the `XGrabKeyboard` function to take control of the keyboard. When a client has called this function, the X server directs all keyboard events only to that client. Using this call, an attacker could easily grab the input stream from a window and direct it to another window. The attacker could return simulated keystrokes to the window to fool the user running the window. Thus, the user might not realize that anything was wrong.

The ability of an attacker to capture a user's keystrokes threatens the confidentiality of the data stored on the workstation.

DECterm windows provide a secure keyboard mode that directs everything a user types at the workstation keyboard to a single, secure window. Users can set this mode by selecting the Secure Keyboard item from the Commands menu in a DECterm window.

Include a secure keyboard mode in programs that deal with sensitive data. This precaution is especially important if your program prompts a user for a password.

Some guidelines for implementing secure keyboard mode follow:

- Use the XGrabKeyboard call to the Xlib library.
- Use a visual cue to let the user know that secure keyboard mode has been set, for example, reverse video on the screen.
- Use the XUngrabKeyboard function to release the keyboard grab when the user reduces the window to an icon. Releasing the keyboard frees the user to direct keystrokes to another window.

7.10.3 Block Keyboard and Mouse Events

Hosts listed in the access control list can send events to any window if they know its ID. The XSendEvent call enables the calling application to send keyboard or mouse events to the specified window. An attacker could use this call to send potentially destructive data to a window. For example, this data could execute the rm -rf * command or use a text editor to change the contents of a sensitive file. If the terminal was idle, a user might not notice these commands being executed.

The ability of an attacker to send potentially destructive data to a workstation window threatens the integrity of the data stored on the workstation.

DECterm windows block keyboard and mouse events sent from another client if the allowSendEvents resource is set to False in the .Xdefaults file.

You can write programs that block events sent from other clients. The XSendEvent call sends an event to the specified window and sets the send_event flag in the event structure to True. Test this flag for each keyboard and mouse event that your program accepts. If the flag is set to False, the event was initiated by the keyboard and is safe to accept.

7.10.4 Protect Device-Related Events

Device-related events, such as keyboard and mouse events, propagate upward from the source window to ancestor windows until one of the following conditions is met:

- A client selects the event for a window by setting its event mask
- A client rejects the event by including that event in the do-not-propagate mask

You can use the XReparentWindow function to change the parent of a window. This call changes a window's parent to another window on the same screen. All you need to know to change a window's parent is the window ID. With the window ID of the child, you can easily discover the window ID of its parent.

The misuse of the XReparentWindow call can threaten security in a windowing system. The new parent window can select any event that the child window does not select.

Take these precautions to protect against this type of abuse:

- Have the child window select the device events that it needs. This precaution prevents the new parent from intercepting events that propagated upward from the child. Parent windows that centralize event handling for child windows are at greater security risk. An attacker can change the parent and intercept the events intended for the children. Therefore, it is safer for each child window to handle its own device events. Events that the child explicitly selects never propagate.

- Have the child window specify that device events will not propagate further in the window hierarchy. This precaution prevents any device event from propagating to the parent window, regardless of whether the child requested the event.
- Have the child window ask to be notified when its parent window is changed by setting the `StructureNotify` or `SubstructureNotify` bit in the child window's event mask. For information on setting these event masks, see the *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD.*

7.11 System Calls and Library Routines with Security Implications

The following tables list many of the ULTRIX system calls and library routines that have security implications for programmers.

ULTRIX C programs can be compiled for BSD, SYSTEM_FIVE, POSIX, or X/Open environments. For detailed information, see the *Reference Pages Section 2: System Calls* and the *Reference Pages Section 3: Library Routines.*

Some system calls and library routines that are not covered in this section might have implicit security concerns. Also, the misuse of a system call or library routine that does not seem to have any security concerns could threaten the security of a computer system. For example, all system calls bypass file access permissions when called by a privileged process. Ultimately, programmers are responsible for the security implications of their programs.

7.11.1 System Calls

Table 7-2 lists the system calls that have security relevance for programmers.

Table 7-2: Security-Relevant System Calls

Category	System Calls	
File control	`creat`	`open`
	`fcntl`	`read`
	`mknod`[a]	`write`
Process control	`fork`	`sigpause`
	`execve`	`sigsetmask`
	`setpgrp`[a]	`sigvec`
	`sigblock`	
File attributes	`access`	`chroot`[a]
	`chmod`[a]	`stat`
	`chown`[a]	`umask`
User and group ID	`getegid`	`getuid`
	`getgid`	`setgroups`[a]
	`geteuid`	`setreuid`[a]
	`setreuid`[a]	
Auditing	`audcntl`[a]	`audgen`[a]
General	`syscall`	

Table note:

a. Either these system calls can be called only by a privileged process or when they are called by a privileged process, they behave differently from the way they behave when called by a nonprivileged process. See the *ULTRIX Reference Pages* for more information.

7.11.2 Library Routines

Library routines are system services that programs can call. Many library routines use system calls. Table 7-3 lists ULTRIX library routines that have security implications.

Table 7-3: Security-Relevant Library Routines

Category	Library Routines	
File control	`fopen`	`popen`

Password handling	`getpass`
	`getpwnam`
	`getpwent`
	`getpwuid`
		`putpwent`
		`setpwent`
		`endpwent`

Process control	`signal`	

Group processing	`getgrent`
	`getgrnam`
	`getgrgid`
		`setgrent`
		`endgrent`

Identifying the user	`cuserid` `getlogin`	`getpwuid`
Password encryption	`crypt`	`encrypt`

User and group ID	`setuid`
	`setegid`
	`seteuid`
		`setgid`
		`setrgid`
		`setruid`

Authorization	`getauthent`
	`getauthuid`
	`storeauthent`
	`authenticate_user`
		`setauthent`
		`setauthfile`
		`endauthent`

Calling Between C and Pascal 8

This chapter describes the coding interfaces between C and Pascal and gives rules and examples for calling and passing arguments between these languages. The chapter addresses the differences for both the RISC and VAX architectures.

For detailed information on how C variables appear in storage when compiled using the `cc` command, see Appendix B. For information about how variables of other languages appear in storage, see the documentation for that language.

8.1 Differences Between C and Pascal

In general, calling C from Pascal and Pascal from C is fairly simple. Both Pascal and C allow only one main routine in a program, which can be written in either Pascal or C. Most data types in each language have natural counterparts in the other language. However, differences do exist in the following areas:

- Passing string data
- Calling routines with a variable number of arguments
- Type checking
- Passing arrays
- Passing single-precision floating point values (VAX specific)
- Passing floating point values (RISC specific)
- Using procedure and function arguments (RISC specific)
- Passing file variables (RISC specific)

8.1.1 Passing String Data

The C and Pascal languages handle strings differently. Pascal handles string data as fixed-length arrays of characters. String parameters are typically declared as follows:

```
VAR S: PACKED ARRAY[1..100] OF CHAR;
```

The upper bound (100 in this case) is assumed to be large enough to handle most processing requirements efficiently. In passing an array, Pascal passes the entire array, as specified, and pads to the end of the array with spaces. Most C functions treat strings as pointers to a single character and use pointer arithmetic to step through the string. A null character (\0 in C) terminates a string in C. Therefore, when passing a string from Pascal to C, terminate the string with a null character (chr(0) in Pascal).

Example 8-1 and Example 8-2 show a Pascal program that calls the `atoi` C function and passes the string `s`. Note that the program ensures that the string terminates with a null character. Example 8-1 shows the program on a RISC system and Example

8-2 shows the program on a VAX system.

Example 8-1: Passing String Data on a RISC System

```
type
  astrindex = 1 .. 20;
  astring = packed array [astrindex] of char;
  function atoi(var c:  astring): integer; external;
program ptest(output);
  var
    s: astring;
    i: astrindex;
  begin
  argv(1, s); { This predefined Pascal function
                is an extension }
  writeln(output, s);
  { Guarantee that the string is null-terminated
    (but may eliminate the last character if the argument
    is too long). "lbound" and "hbound" are extensions. }
  s[hbound(s)] := chr(0);
  for i := lbound(s) to  hbound(s) do
    if s[i] = ' ' then
             begin
             s[i] := chr(0);
             break;
             end;
  writeln(output, atoi(s));
  end.
```

Example 8-2: Passing String Data on a VAX System

```
program example(output);
type
    examplestr = packed array [1..10] of char;
var
    s : examplestr;
    i : integer;
function atoi( var s : examplestr ) : integer; external;
begin
    s := '100';
    s[4] := chr(0);
    i := atoi(s);
    writeln(i);
end.
```

For more information on `atoi`, see `atof`(3) in the *ULTRIX Reference Pages.*

8.1.2 Calling Routines with a Variable Number of Arguments

You can define C functions that take a variable number of arguments (for example, `printf` and its variants). Such functions can be called from Pascal, but they must be defined with a specific number of parameters in your Pascal program.

8.1.3 Type Checking

Pascal performs run-time checks on certain variables for errors; in contrast, C does not. For example, when a reference to an array exceeds its bounds in a Pascal program, the error is flagged (if run-time checks are not suppressed). Do not expect a C function to detect similar errors when you pass data to it from a Pascal program.

8.1.4 Passing Arrays

C never passes arrays by value. In C, an array is actually a pointer. Therefore, passing an array actually passes its address, which corresponds to Pascal variable-parameter (VAR) array passing, or passing by reference. Passing Pascal arrays by reference (VAR) instead of value is usually more efficient. Therefore, most Pascal array parameters are VARs. When it is necessary to call a Pascal routine with a by-value array parameter from C, pass a C structure containing the corresponding array declaration.

8.1.5 Passing Single-Precision Floating Point Values (VAX Specific)

Pascal on the VAX platform provides only double-precision floating-point data (Pascal data type real). Thus, only double-precision floating-point data can be passed (C data type double).

8.1.6 Passing Floating Point Values (RISC Specific)

In function calls, C automatically converts single-precision floating point values to double precision. By contrast, Pascal passes single-precision floating by-value arguments directly.

When passing double-precision values between C and Pascal routines, follow these guidelines:

- If possible, write the Pascal routine so that it receives and returns double-precision values.
- If the Pascal routine cannot receive a double-precision value, write a Pascal routine to accept double-precision values from C and then have that routine call the single-precision Pascal routine.

Passing single-precision values by reference between C and Pascal does not pose a problem.

8.1.7 Using Procedure and Function Arguments (RISC Specific)

C function variables and arguments consist of a single pointer. In contrast, Pascal procedure and function arguments consist of a pointer to machine code and a pointer to the stack frame of the lexical parent of the function. Such values can be declared as structures in C. To create such a structure, put the C function pointer in the first word and zero in the second. C functions cannot be nested and, thus, have no lexical parent. Therefore, the second word is irrelevant.

Note that you cannot call a C function with a function parameter from Pascal.

8.1.8 Passing File Variables (RISC Specific)

The Pascal text type and the C `stdio` package's declaration `FILE*` are compatible. However, Pascal passes file variables only by reference; a Pascal routine cannot pass a file variable by value to a C function. As with any reference parameter, C functions that pass files to Pascal routines should pass the address of the `FILE*` variable.

8.2 Calling Pascal from C

To call a Pascal routine from a C program, follow these steps:

1. Write a C extern declaration in the following form:

   extern void *name*();

2. Call the Pascal procedure with actual arguments.

   Be sure that the arguments are of the type that Pascal expects. Table 8-1 lists the C argument types that match those expected by the called Pascal routine.

Table 8-1: C Argument Types

Pascal Type Expected	C Type
integer	integer or char value -2^{31} .. 2^{31} -1
subrange	integer or char value in subrange
char	integer or unsigned char (0 to 127)
boolean	integer or char (0 or 1)
enumeration	integer or char (0 .. N−1)
pointer types	pointer type und <0. := lbound(s)
reference parameter	pointer to the appropriate type
record types	structure or union type
by-reference array parameters	corresponding array type
by-value array parameters	structure that contains the corresponding array
	RISC Specific
cardinal	unsigned int
real	none
double	float or double
procedure	struct *{void *p(); int *l}*
function	struct *{function-type *f(); int *l}*
by-reference text	FILE**
	VAX Specific
real	double

3. Declare a variable in which to store the return value.

   Be sure the return value data type and the data type of the variable you declare are compatible. Table 8-2 provides guidelines for declaring a return value type.

Table 8-2: Guidelines for Declaring Return Value Types

Pascal Return Value Type	C Type Declaration
integer[a]	int
char	char
boolean	char
enumeration	unsigned or corresponding enum (signed in C)
pointer type	corresponding pointer type
record type	corresponding structure or union type
array type	structure containing corresponding array type
none	void
	RISC Specific
cardinal[b]	unsigned int
real	none
double	double
	VAX Specific
real	double

Table notes:

a. Applies also to subranges with lower bounds <0.

b. Applies also to subranges with lower bounds ≥0.

8.2.1 Calling Pascal from C on a RISC System

This section contains examples of calling a Pascal program from a C program on a RISC system.

To pass a pointer to a function in a call from C to Pascal, you must pass a structure by value. The first word of the structure must contain the function pointer, and the second must contain a zero. Pascal requires this format because it expects an environment specification in the second word.

Example 8-3 shows code for a C function calling a Pascal function.

Example 8-3: Calling a Pascal Function

Pascal function:
```
function bah(
     var f: text;
     i: integer
     ): double;
  begin
     .
     .
     .
  end {bah};
```

Example 8-3: (continued)

C declaration of bah:
```
extern double bah();
```
C call:
```
int i; double d;
FILE *f;
d = bah(&f, i);
```

Example 8-4 shows a C function calling a Pascal procedure.

Example 8-4: Calling a Pascal Procedure

Pascal procedure:
```
type
  int_array = array[1..100] of integer;
procedure zero (
    var a: int_array;
    n: integer
    )
   begin
     .
     .
     .
  end {zero};
```
C declaration:
```
extern void zero();
```
C call:
```
int a[100]; int n;
zero(a, n);
```

Example 8-5 shows a C function that passes strings to a Pascal procedure, which then prints them. Note the following:

- The Pascal procedure must check for the null [chr(0)] character, which indicates the end of the string passed by the C routine.
- The Pascal procedure must not write to output; instead, it uses the stdout file-stream descriptor passed by the C routine.

Example 8-5: Passing a String to a Pascal Procedure (RISC Specific)

C call:
```
/* Send the last command-line argument to Pascal routine */
#include <stdio.h>
main(argc, argv)
  int argc; char **argv;
  {
  FILE *temp = stdout;
  if (argc != 0)
    p_routine(&temp, argv[argc - 1]);
  }
```
Pascal procedure:
```
{ We assume the string passed to us by the routine
  will not exceed 100 bytes in length }
type
  astring = packed array [1..100] of char;
procedure p_routine(var f: text; var c: astring);
  var
    i: integer;
  begin
  i := lbound(c);
  while (i < hbound(c)) and (c[i] <> chr(0)) do
    begin
    write(f, c[i]);
    i := i + 1;
    end;
  writeln(f);
  end;
```

8.2.2 Calling Pascal from C on a VAX System

Example 8-6 is a C to Pascal call on a VAX system. It shows a C function that passes strings to a Pascal procedure, which then prints them. Note the following:

- The Pascal procedure must check for the null [chr(0)] character, which indicates the end of the string passed by the C routine.
- The Pascal procedure must not write to output; instead, it uses the stdout file-stream descriptor passed by the C routine.

Example 8-6: Passing a String to a Pascal Procedure (VAX Specific)

C call:

```
/* Send the last command-line argument to Pascal routine */
#include <stdio.h>
main(argc, argv)
  int argc; char **argv;
  {
  if (argc != 0)
    p_routine(argv[argc - 1]);
  }
```

Pascal procedure:

```
{ We assume the string passed to us by the routine
  will not exceed 100 bytes in length }
type
  astring = packed array [1..100] of char;
procedure p_routine(var c: astring);
  var
    i: integer;
  begin
  i := 1;
  while (i < 100) and (c[i] <> chr(0)) do
    begin
    write(c[i]);
    i := i + 1;
    end;
  writeln;
  end;
```

8.3 Calling C from Pascal

Follow these steps to call a C function from Pascal:

1. Write a Pascal declaration that describes the C function.

   If the C function returns a value, write a Pascal function declaration. Otherwise, you can write a Pascal procedure declaration.

2. Write the call, specifying the actual arguments you want to pass.

   Be sure that the data types of the arguments are data types that the C function expects. Table 8-3 describes the Pascal argument types that match those expected by the called C function.

Note that on RISC systems, a Pascal routine cannot pass a function pointer to a C function.

Table 8-3: Pascal Argument Types

Type Expected By C Function	Pascal Type
int[a]	integer
short[b]	integer (or –32768 .. 32767)
unsigned short	cardinal (or 0 .. 65535)
unsigned char	char
char[c]	integer (or –128 .. 127)
enum type	corresponding enumeration type
struct type	corresponding record type
union type	corresponding record type
array type	corresponding array type passed by reference (VAR)
	RISC Specific
unsigned int[d]	cardinal
unsigned short	cardinal (or 0 .. 65535)
float	double
double	double
FILE *	text (passed by reference – VAR)
FILE **	corresponding pointer type or corresponding type passed by reference (VAR)
	VAX Specific
unsigned int [c]	none
unsigned short	0 .. 65535
double	real

Table notes:

a. Same as types signed int, long, signed long, signed.

b. Same as type signed short.

c. Same as type signed char.

d. Same as types unsigned, unsigned long.

3. Declare a return value for the function, if necessary.

Portable C (pcc) Implementation Notes A

The C language supported by the pcc ULTRIX compiler is an implementation of the language defined in *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978). The language that the pcc compiler supports differs from the C defined by Kernighan and Ritchie in certain ways. This appendix discusses the pcc language implementation details.

A.1 Specifying the varargs.h Macros

If a function takes a variable number of arguments (for example, the C library functions printf and scanf), you must use the macros defined in the varargs.h header file.

The va_dcl macro declares the formal parameter va_alist, which is either the format descriptor for the remaining parameters or a parameter itself.

The va_start macro must be called within the body of the function whose argument list is to be traversed. The function then can transverse the list or pass its va_list pointer to other functions to transverse the list. The type of the va_start argument is va_list, which is defined in varargs.h.

The va_arg macro accesses the value of an argument rather than obtaining its address. This macro handles those type names that can be transformed into the appropriate pointer type by appending an asterisk (*), which handles most simple cases.

The argument type in a variable argument list must never be an integer type smaller than int and must never be float.

For more information, see varargs(5) in the *ULTRIX Reference Pages.*

The following example illustrates using varargs macros:

```
#include <varargs.h>
#include <stdio.h>
enum operations {load, store, add, sub};
main() {
  void emit();
  emit(load, 'I',  0,  4);
  emit(load, 'I',  4,  4);
  emit(add,  'I');
  emit(store,'I',  0,  4);
}
void
emit(op, va_alist)
/* emit takes a variable number of arguments and prints
/* them according to the operational format. */
enum operations op;
va_dcl {
va_list arg_ptr;
register int length, offset;
register char type;
va_start(arg_ptr);
switch(op) {
    case add:  /* print operation and length */
      type = va_arg(arg_ptr, int);
      printf("add %c\n", type);
      break;
    case sub:  /* print operation and length */
      type = va_arg(arg_ptr, int);
      printf("sub %c\n", type);
      break;
    case load: /* print operation, offset, and length */
      type = va_arg(arg_ptr, int);
      offset = va_arg(arg_ptr, int);
      length = va_arg(arg_ptr, int);
      printf("load %c %d %d\n", type, offset, length);
      break;
    case store:
      type = va_arg(arg_ptr, int);
      offset = va_arg(arg_ptr, int);
      length = va_arg(arg_ptr, int);
      printf("store %c %d %d\n", type, offset, length);
  }
}
```

The expected output from this code is as follows:

```
load I 0 4
load I 4 4
add I
store I 0 4
```

A.2 Deviations

C does not support the `entry` keyword, which has no defined use. Additionally, on the RISC architecture C does not support the `asm` keyword as implemented by some C compilers to allow for the inclusion of assembly language instructions.

A.3 Extensions

ULTRIX language extensions to Kernighan and Ritchie C include the following:

- The `enum` type is a set of values represented by identifiers called enumeration constants; enumeration constants are specified when the type is defined. For information on the alignment, size, and value ranges of the `enum` type, see Appendix B.
- The `void` type allows you to specify that no value be returned from a function.
- The `volatile` type modifier is used when programming I/O devices, In addition, the `const` keyword has been reserved for future use. For more information on the `volatile` modifier, see Chapter 2.

A.4 Translation Limits

Table A-1 lists the maximum limits imposed on certain items by the `pcc` compiler:

Table A-1: C Compiler Limitations

C Specifications	Maximum
Nesting Levels	
Compound statements	≤30
Iterations	
Selections	
Conditional compilations	
Maximum number of type modifiers (array, pointers, function, volatile)	9
Case labels	500
Function call parameters	150

Storage Mapping for C Data B

This appendix describes the alignment, size, and value ranges for C data types. The appendix also describes the alignment the compiler uses for arrays, structures, and unions. Finally, the appendix describes the C storage classes. Except where differences are noted, the information in this appendix applies to both the RISC and VAX architectures.

For information on the storage mapping for languages other than C, refer to the compiler documentation for that language.

B.1 Alignment, Size, and Value Ranges of C Data Types

Table B-1 describes how the C compiler implements size, alignment, and value ranges for each data type.

Table B-1: C Data Type Size, Alignment, and Value Ranges

Type	Size	Alignment	Signed	Unsigned
int	32 bits	word[a]	-2^{31} to 2^{31} -1	0 to 2^{32} –1
long	32 bits	word[a]	-2^{31} to 2^{31} –1	0 to 2^{32} –1
enum	32 bits	word[a]	-2^{31} to 2^{31} –1	
short	16 bits	halfword[a]	–32,768 to 32,767	0 to 65,535
char[b]	8 bits	byte	–128 to 127	0 to 255
float[c]	32 bits	word[a]	See Table B-2	
double[d]	64 bits	doubleword[e]	See Table B-2	
pointer	32 bits	word[a]	0 to 2^{32} –1	

Table notes:

a. Byte boundary divisible by 4.

b. Byte boundary divisible by 2.

c. Unless the unsigned attribute is used, char is assumed to be signed.

d. Single precision floating point (IEEE single precision on RISC and F-floating on VAX).

e. Double precision floating point (IEEE double precision on RISC and D- or G-floating on VAX).

f. Byte boundary divisible by 8.

Table B-2 shows the approximate valid value ranges for float and double data types.

Table B-2: Size Ranges for the Float and Double Data Types

	Float	Double
RISC Specific		
Maximum Value	$3.40282356 * 10^{38}$	$1.7976931348623158 * 10^{308}$
Normalized Minimum Value	$1.17549429 * 10^{-38}$	$2.2250738585072012 * 10^{-308}$
Denormalized Minimum Value	$1.40129846 * 10^{-46}$	$4.9406564584124654 * 10^{-324}$
VAX Specific		
Maximum Value	$1.7014118 * 10^{38}$	$1.70141183460469229 1 * 10^{38}$ (D-float) $8.988465674311579 * 10^{307}$ (G-float)
Minimum Value	$2.9387359 * 10^{-39}$	$2.93873587705571880 * 10^{-39}$ (D-float) $5.5626846462680035 * 10^{-309}$ (G-float)

The `limits.h` and `float.h` header files, which are usually found in `/usr/include`, contain C macros that define minimum and maximum values for the various data types. For information about the macro names and values, see the appropriate header file.

B.2 Storage Mapping of C Arrays, Structures, and Unions

An array has the same boundary requirements as the data type specified for the array. The size of an array is the size of the data type multiplied by the number of elements. For example:

```
double x[2][3]
```

The size of the resulting array would be 48 bytes (that is, 2*3*8, where 8 is the size in bytes of the double floating-point type).

Each member of a structure begins at an offset from the structure base. The offset corresponds to the order in which a member is declared; the first member is at offset 0.

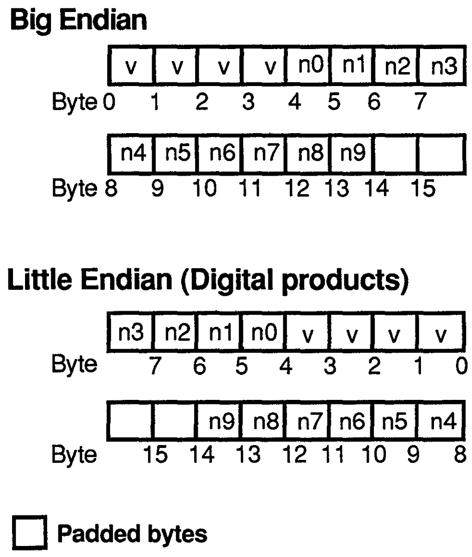
The size of a structure in the object file is the size of its combined members plus padding added, where necessary, by the compiler. The following rules apply to structures:

- Structures must align on the same boundary as that required by the member with the most restrictive boundary requirement. The boundary requirements, by increasing degree of restrictiveness, are byte, halfword, word, and doubleword.
- The compiler ends the structure on the same alignment boundary on which it begins. For example, if a structure begins on an even-byte boundary, it also ends on an even-byte boundary.

The following example shows a structure declaration:

```
struct S {
     int v;
     char n[10];
}
```

The following figure illustrates how this structure would exist when mapped out in storage:



ZK-0065U-R

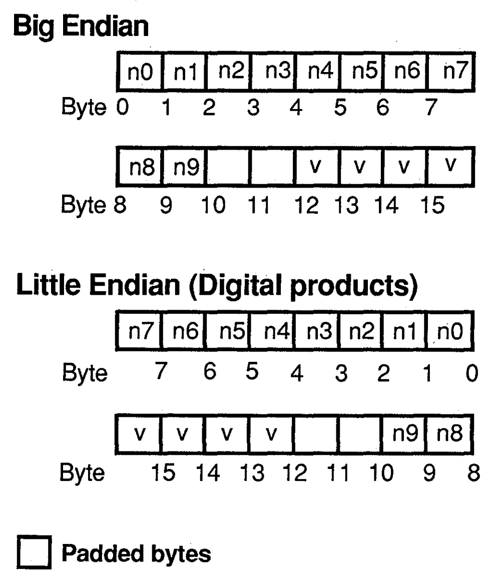
Even though the byte count defined by the `int v` and `char n` components is 14, the length of the structure is 16 bytes. Because `int` has a stricter boundary requirement (word boundary) than char (byte boundary), the structure must end on a word boundary (a byte offset divisible by 4). Therefore, the compiler adds two bytes of padding to meet this requirement.

An array of data structures illustrates the reason for this requirement. For example, if the structure in the previous figure were the element type of an array, some of the `int v` components would not be aligned properly without the 2-byte pad.

Alignment requirements may cause padding to appear in the middle of a structure. For example:

```
struct S {
     char n[10];
     int v;
}
```

The following figure illustrates how this structure would exist when mapped out in storage:



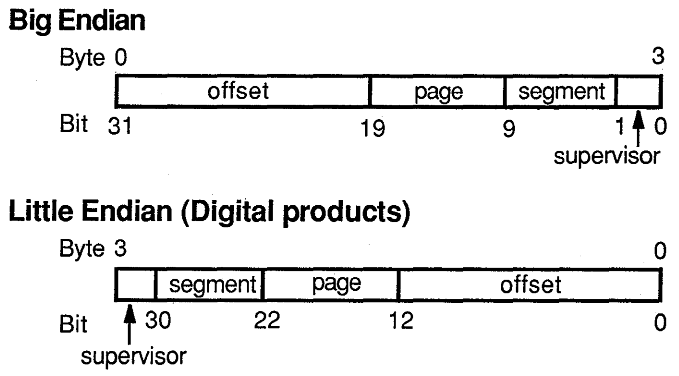
ZK–0066U–R

Note that the size of the structure remains 16 bytes, but two bytes of padding follow the n component to align v on a word boundary.

Bit fields are packed from the most-significant bit to least-significant bit in a word; they cannot exceed 32 bits; and they can be signed or unsigned. For example:

```
struct S {
     unsigned offset :12;
     unsigned page :10;
     unsigned segment :9;
     unsigned supervisor :1;
}virtual_address;
```

The following figure illustrates how this structure would exist when mapped out in storage:



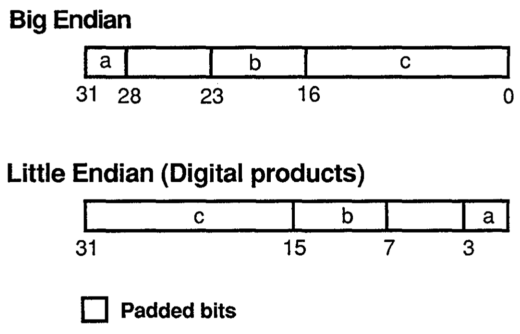
ZK–0067U–R

Note that the compiler moves the fields that overlap a word boundary to the next word.

The compiler aligns a nonbit field that follows a bit-field declaration to the next boundary appropriate for its type. For example:

```
struct S {
     unsigned a :3;
     char b;
     short c;
}x;
```

The following figure illustrates how this structure would exist when mapped out in storage:



ZK-0068U-R

Note that five bits of padding are added after `unsigned a` so that `char b` aligns on a byte boundary, as required.

A union must align on the same boundary as the member with the most restrictive boundary requirement. The boundary requirements, by increasing degree of restrictiveness, are: byte, halfword, word, and doubleword. For example, a union containing char, int, and double data types must align on a doubleword boundary, as is required by the double data type.

B.3 C Storage Classes

Table B-3 lists the C storage classes.

Table B-3: C Storage Classes

Class	Description
`auto`	Storage is allocated at execution and exists only for the duration of that block activation.
`static`	The compiler allocates storage, which remains fixed for the duration of the program. Static variables reside in the program's bss section if they are not initialized; otherwise, they are placed in the data section.
`register`	The compiler allocates variables with the `register` storage class to registers. For programs compiled with the `-O` option, the optimization phase of the compiler tries to assign all variables to registers, regardless of the storage class specified.

Table B-3: (continued)

Class	Description
`extern`	The variable refers to storage defined elsewhere in an external data definition. The compiler does not allocate storage to `extern` variable declarations; it uses the following logic in defining and referencing them: If you omit `extern` and an initializer is present, a definition for the symbol is emitted. If you specify two or more such definitions among all the files that form a program you receive an error message at link time or before. If no initializer is present, a common definition is emitted. Any number of common definitions of the same identifier can coexist. If you specify `extern` the compiler assumes that declaration refers to a name defined elsewhere. A declaration having an initializer is invalid. If you never use an identifier you declare, the compiler does not issue an external reference to the linker.

B.4 volatile Type Qualifier (RISC Specific)

You specify the `volatile` type qualifier for variables that may be modified in ways unknown to the compiler. For example, you might specify `volatile` for an object corresponding to a memory mapped input/output port or an object accessed by an asynchronously interrupting function. Except for expression evaluation, no phase of the compiler optimizes any of the code dealing with objects declared as `volatile`.

If you assign a volatile pointer to another pointer without the `volatile` specification, the compiler treats the other pointer as nonvolatile. For example, suppose a program contains the following declarations and assignment statement:

```
volatile int *i;
int *j;
   .
   .
   .
j = i;
```

The compiler treats the assignment statement as if the pointer `j` has been cast as follows:

```
(volatile*)j = i
```

The compiler treats `j` as a nonvolatile pointer and the object it points to as nonvolatile (the compiler may optimize it). Note that the `-volatile` compiler option causes all objects to be compiled as volatile.

Porting Applications from a VAX System to a RISC System C

Many of the applications you write to run on a VAX ULTRIX system will run with little modification on a RISC ULTRIX system. Other applications will need substantial modification for you to port them from a VAX to a RISC system.

This appendix describes differences between the VAX and RISC systems. The appendix also gives information on modifying your VAX program so that it will run on a RISC system.

C.1 Differences in Files

One of the differences between RISC and VAX systems is the size, format, and contents of files. The following sections describe these differences in files.

C.1.1 Executable Image Size

Executable images on RISC systems are larger and therefore take up more disk space than their counterparts on VAX systems. This size difference is due to the instruction set of the RISC architecture. Typically, images on RISC systems are 30 to 40 percent larger than on VAX systems.

C.1.2 Object Format

Unlike VAX systems, RISC systems use Common Object File Format (COFF) for object files. If your program contains hard-coded initializations that depend upon the VAX `nlist` structure, you must change them when you port the program to a RISC system.

C.1.3 Contents of the a.out.h File

The `a.out.h` header file does not include `exec.h` on RISC systems, as it does on VAX systems.

C.2 Differences in Functions

Some system calls or other functions have a different effect or are used differently on a RISC system than on a VAX system. This section describes differences in functions.

C.2.1 The brk and getrlimit System Calls

On VAX systems, virtual address space begins at zero. Program text starts at zero and runs to `&etext`. Program data begins after `&etext` and follows to `&edata`. The bss segment then follows to `&end`, and the rest of memory is available for growth.

On RISC systems, the virtual address space begins at 0x00400000. The text segment starts at 0x00400000 and runs to `&etext`. Rather than beginning directly after `&etext`, data begins at 0x10000000. The data segment continues to `&edata` and is followed by the bss segment to `&end`. The rest of memory is available for growth.

This difference changes the interaction between the `brk` and `getrlimit` system calls. On VAX systems, when you call `getrlimit` to get RLIMIT_DATA, the value returned by `getrlimit` is an approximation for the maximum value that you could pass to the `brk` system call. On RISC systems, the correct value is as follows:

```
"the value returned by a getrlimit" + 0x10000000
```

One way to work around this problem is to use the `sbrk` call, instead of the `brk` call.

C.2.2 Functions that Return a Pointer

On VAX systems, if a function that returns a pointer returns –1 error status, you can make the following comparison:

```
if (ptr < 0)
```

This comparison can be true because pointers are signed values on a VAX system. On a RISC system, the comparison is never true because pointers are unsigned values. The compiler removes the code for the comparison, so the program cannot catch the error status.

On a RISC system, you can use the following comparison to test the return value when a function returns a pointer:

```
if ((int)ptr < 0)
```

The following comparison is also valid:

```
if (ptr == (char *) -1)
```

C.3 Attach Point for Shared Memory Segments

The attach points for shared memory segments in the virtual address space of a process on a RISC system are different from a process on a VAX system. On both systems, you attach shared memory segments by using the `shmat` system call. On a RISC system, the shared memory segments fall between the text segment and the private data segment, by default. Therefore, you can expand your private data segment (by using the `sbrk` or `brk` system call) regardless of an attached shared memory segment.

When you create a shared memory segment on a RISC system, its attach point in the virtual address space of your process must be aligned on a 4-megabyte boundary. If you let the system default create the attach point, the system aligns the shared memory segment properly. If you must explicitly attach to a given address, that address must be at a 4-megabyte boundary or you must set the SHM_RND flag. If the SHM_RND flag is set, the system rounds the address you specify to a 4-megabyte boundary. This restriction is imposed by hardware constraints.

Whenever possible, use the system default to create an attach point. For details on creating attach points, see `shmop(2)` in the *ULTRIX Reference Pages*.

C.4 Differences in Data Representation and Manipulation

RISC and VAX systems occasionally differ in the way they represent and manipulate data. The following sections describe these differences.

C.4.1 Floating Point and Double Precision Data

Unlike VAX systems, the kernel on a RISC system does not manipulate floating point or double precision values in the kernel. The kernel manipulates only integer values. The kernel assigns the Floating Point Unit (FPU) to a process.

The `fixpoint.h` header file contains macros to convert an integer to its floating point format. You can include this header file in your program and use the macros.

In addition, VAX processors typically use D-float floating-point format. RISC processors use IEEE floating-point format, which is similar to G-float format. Thus, on RISC systems, you can use a greater range of floating point numbers but the numbers have less precision (fewer decimal places).

If your program needs the extra precision of D-float format or if your program must be cognizant of the low-level format of floating point numbers, it will be difficult to port to a RISC system. RISC systems provide no equivalent to D-float or H-float format.

If your program incorrectly treats a floating value as a double precision value, or a double precision value as floating, it might run on a VAX system, in spite of the error. On a RISC system, this programming error causes incorrect results.

C.4.2 NULL Pointers

On VAX systems, you can dereference a NULL pointer because page zero of user process space is mapped and valid. On RISC systems, however, you cannot dereference a NULL pointer without a segmentation violation. You must test the pointer to be sure it is not NULL before you dereference it.

C.4.3 Data Alignment

On VAX systems, short words (2 bytes) or long words (4 bytes) can be accessed on any byte boundary. On RISC systems, however, references must be ‘‘naturally’’ aligned. Short words (2 bytes) must be on an even byte boundary. Long words (4 bytes) must be accessed on a boundary evenly divisible by 4.

If your program contains an unaligned access, the system attempts to correct the unaligned access. If the system is able to accomplish the correction, it displays a message on the controlling terminal (if one exists) stating at what pc the alignment error was encountered. If the system is not able to correct the unaligned access, it terminates the process with a SIGBUS (bus error) signal. The correction that the system attempts might affect the performance of your program.

One common cause of an attempt to access unaligned data occurs when you use the C language. Although the C compiler aligns data based on its size, it allows you to create a pointer to unaligned data by casting a variable. For example, suppose your program contains a pointer to `char`, which you cast to be a pointer to `struct`.

The alignment rules for a structure are more restrictive than they are for a character string. Because of the alignment rules, the pointer to `struct` can create an unaligned access to the `char *` buffer if you use it to access the middle of the buffer.

For further information, see `uac`(1) in the *ULTRIX Reference Pages*.

C.5 Page Size

The page size on a RISC system is 4 kilobytes (4*1024 bytes). On a VAX system, the page size is 512 bytes and ULTRIX manipulates data two pages at a time (or 1 kilobyte [2*512 bytes] at a time). This page size difference can affect the memory management performed by your program.

If your program manipulates memory using pages, get the page size by using the `getpagesize` system call. (For further information, see `getpagesize`(2) in the *ULTRIX Reference Pages*.) Alternatively, you can include the `vmmac.h` header file and use the macros defined in it for page size manipulations.

C.6 Command Differences

Some commands that you use during program development differ between RISC and VAX systems. This section describes the differences in the commands.

C.6.1 The prof Command

The `prof` command provides information about what routines in your program execute the most or the least. The RISC version of `prof` is functionally different from the VAX version. For an explanation of using `prof` on a RISC system see Section 4.3.

C.6.2 The ranlib Command

On a VAX system, the `ranlib` command organizes archives of object files to allow faster linking. This command exists on RISC systems as a shell script that passes a flag to the `ar` librarian. The `ar` librarian performs the same function as `ranlib`.

C.6.3 The lint Command

The `lint` command searches your program for coding errors, coding that is not portable, and inefficient coding. The `lint` command differs on RISC and VAX systems. The differences are in the messages `lint` displays, the conditions it checks, and the command you use to build libraries. On a VAX system, you use the following command to build a `lint` library for a program named `myprog.c`:

```
% lint -C libname myprog.c
```

On a RISC system, you use the following command:

```
% lint -c myprog.c
```

For more information about `lint`, see Section 4.1.

C.6.4 Commands that Read or Write Object Files

RISC and VAX systems use different formats for object files and load modules. Therefore, the following utilities are slightly different on the two architectures:

- `ar`
- `dbx`
- `ld`
- `nm`
- `size`
- `strip`

C.7 C Compiler Differences

The RISC C (`cc`) compiler is different from the VAX C (`cc`) compiler. The following list describes differences between the two compilers:

- The RISC C compiler does not support the `const` keyword.
- Pointers on RISC systems are unsigned; on VAX systems they are signed.
- You cannot use a pointer as the expression with a `switch` statement on RISC systems.
- On RISC systems, you cannot dereference NULL pointers, including arguments to the `strlen` function.
- The RISC C compiler does not support the `asm` pseudo function call.
- The RISC C compiler does not allow the following obsolete form of initialization:

  ```
  int i 0;
  ```

  The preceding example works on a VAX system, but the VAX C compiler issues a warning. The example generates an error message on a RISC system.
- The RISC C compiler has boundary alignment rules. The only effect this difference should have on your program is that its performance might be slower than on a VAX system. This performance change could occur because the kernel corrects alignment errors. Where possible align double-words, words, and half-words on ``natural'' boundaries. For more information, see Section C.4.3 and `uac`(1) in the *ULTRIX Reference Pages*.
- The `varargs` function is different on RISC systems. Your program will fail if it ``walks'' an argument list by incrementing the address of an argument, particularly if the arguments are double precision values. Use the macros in `varargs.h` when you use functions that have a variable number of arguments. Compiling with the `-varargs` option on RISC systems causes the compiler to detect nonportable code.
- The `setjmp/longjmp` buffer is larger on RISC systems than on VAX systems. Programs with a hard-coded 10 word buffer fail; programs that include `setjmp.h` and declare a variable of type `jmp_buf` work correctly.
- On RISC systems, global symbols do not have an extra leading underscore. This difference mostly affects assembly-language programs.

- RISC systems define a macro (for example, LANGUAGE_C) for the preprocessor that makes it possible to write multilingual include files.
- For `cpp` predefined symbols, `ultrix`, `unix`, and `bsd4_2` are defined on both RISC and VAX systems. On RISC systems, the equivalent predefined symbol of `vax` is `mips`. RISC systems also supports the symbols `MIPSEL` and `host_mips`.
- If you use a global data item as if it is a code location (for example, if a data structure has the same name as a system call), the compiler displays an error message similar to the following one at load time:

  ```
  /lib/libc.a(gethostent.o): jump relocation out-of-range, bad object
  file produced, can't jump from 0x4197a0 to 0x10008198 (stat)
  ```

  If you see this message, change the name of the data structure. (In this example, it was named `stat`.)
- The RISC C compiler does not allow the same `.c` or `.o` file to be listed twice on a command line. The compiler generates doubly defined symbol errors. The VAX C compiler allows you to specify the same source or object file twice.
- On VAX systems, the `cc -L` option on the command line affects all `-l` options. On RISC systems, the `cc -L` option operates only on `-l` options that follow it. Therefore, if you want the `-L` option to affect all `-l` options, you must specify the `-L` option first.
- The `-Md` or `-Mg` options are not needed on RISC systems. The hardware has only one double precision format.
- The RISC C compiler does not support the `-R` option (read-only text).
- Profiling on VAX systems has two levels that can be selected with the `-p` and `-pg` options. Profiling on RISC systems also has two levels that can be selected with the `-p` option or by running the post-processor `pixie` program. The RISC C compiler is not affected by either option; all work is done in the assembler or loader (or postprocessor).
- One level of optimization exists on VAX systems, which is off by default and enabled with the `-O` option. Five levels of optimization exist on RISC systems. By default, the second level is used, which can be disabled with the `-O0` option. The `-O` or `-O2` options invoke optimization that is comparable to the optimization on a VAX system. To have you program optimized more fully, use the `-O3` and `-O4` options. The RISC C compiler also has the `-Olimit` option that allows optimization to be bypassed with overly complicated code sections. For more information, see Section 4.5.
- On both RISC and VAX systems, the `-t` and `-B` options specify passes and paths. However, RISC systems provide more pass names. In addition, the RISC C compiler option `-h` is equivalent to the VAX C compiler option `-B`. The `-B` option on RISC systems specifies a suffix for the pass name.
- Like optimization, RISC systems offer four levels for debugging information (controlled by the –g option). VAX systems have only two (on and off).

Kernel Debugging D

This appendix describes how to debug the ULTRIX kernel, /vmunix, using various ULTRIX programs. The debugging procedure differs for RISC and VAX processors; this chapter addresses both processor types.

D.1 RISC Kernel Debugging

This section shows how to debug the ULTRIX kernel, /vmunix, on a RISC system.

Before you debug the kernel, you should understand the following:

- The layout of system memory, which is described in Table D-1
- The layout of the stacks, which is described in Table D-2
- The layout of address space, which is described in Table D-3

Table D-1: System Memory Map

Physical Address	KSEG1	Use
0x00030000	0xa0030000 upward	ULTRIX kernel text, data, and bss
0x0002ffff *to* 0x00020000	0xa002ffff 0xa0020000	Additional PROM space (64K)
0x0001ffff *to* 0x0001fc00	0xa001ffff 0xa001fc00	1K netblock (host and client network boot information)
0x0001fbff *to* 0x0001f800	0xa001fbff 0xa001f800	1K ULTRIX save state area
0x0001f7ff *to* 0x0001f400	0xa001f7ff downward 0xa001f400	1K ULTRIX temporary startup stack
0x0001f3ff	0xa001f3ff downward	dbgmon stack (a few K less than 64K)
0x00010000	0xa0010000 upward	dbgmon text, data, and bss
0x0000ffff	0xa000ffff downward	PROM monitor stack
0x00000500	0xa0000500 upward	PROM monitor bss
0x000004ff *to* 0x00000400	0xa00004ff 0xa0000400	Restart block
0x000003ff *to* 0x00000080	0xa00003ff 0xa0000080	General exception code (note CPU addresses as 0x80000080)
0x0000007f	0xa000007f	

Table D-1: (continued)

Physical Address	KSEG1	Use
to		utlbmiss exception code
`0x00000000`	`0xa0000000`	(note CPU addresses as 0x80000000)

The kernel has no interrupt stack: only kernel, user, and idle stacks.

Table D-2: Stacks on RISC Systems

Stack	Description
Startup stack	Starts at 0x8001 f7ff, growing downward, and is used during system startup until a kernel stack is available
Kernel stack	Starts at 0xffff e000 (KSEG2 space) and grows down
User struct	Starts at 0xffff c000 (KSEG2 space) and goes up
Per-CPU database	Starts at 0xffff 8000 (KSEG2 space) and goes up
User stack	Starts at 0x7fff f000 (KUSEG space, one guard page 0x7fff f000 to 7fff ffff) and grows down

The system is always in virtual address mode; there is no physical address mode.

Table D-3: Address Space on RISC Systems

Address Space	Description
KSEG0	Not mapped, cached—for kernel text Virtual address: 8000 0000 $\rightarrow$ 9fff ffff (512 MB)
KSEG1	Not mapped, not cached—for I/O space Virtual address: a000 0000 $\rightarrow$ bfff ffff (512 MB)
KSEG2	Mapped, cached—for stacks and kernel mallocs Virtual address: c000 0000 $\rightarrow$ ffff ffff (1 GB)
KUSEG	Mapped, cached—for user space Virtual address: 0 $\rightarrow$ 7fff ffff (2 GB)

More information about debugging an ULTRIX kernel on a RISC system can be found in the following header files:

```
/sys/h/proc.h
/sys/h/user.h
/sys/machine/mips/entrypt.h
/sys/machine/mips/frame.h
/sys/machine/mips/pcb.h
/sys/machine/mips/pte.h
/sys/machine/mips/reg.h
```

The `crash` System V program might also be useful.

D.1.1 Using nm to Determine Where a Crash Occurred

When you system crashes, you can use nm to determine which routine was executing when the crash occurred. This tool is most useful when your system does not create a core dump, but displays and Exception Program Counter (EPC) on the console. The following command displays the name list (symbol table) of the vmunix image in numerical order:

```
% nm -n /vmunix
```

To determine which routine was executing, find the address that is closest to, but less than, the EPC from the crash. This address is the starting address of the routine executing when the system crashed. Subtract the start address of this routine from the EPC to get the offset from the beginning of the routine in which the error occurred. Then, use the dbx debugger to find the incorrect instruction.

The following shows an example of nm output:

```
First Kernel text address: 8003,0000 (192k bytes above 8000,0000)
    80030000 T start
    80030000 T eprol
    800300ac T putstr
    80030148 T lputc
    8003018c T cn_reset
        .
        .
        .
First Kernel data address: is approximately 8011,0000
    80112030 D Sysmap
    8011c830 D Usrptmap
    8011f920 D camap
    8011f930 D kmempt
    8011f930 D ecamap
    80123930 D Forkmap
```

D.1.2 Debugging a RISC Kernel with dbx

You can use the dbx debugger to debug a kernel that crashes and creates a core file. You can determine where the crash occurred and use dbx to display instructions and data.

To invoke dbx to debug a nonrunning kernel, issue the following command:

```
% dbx -k vmunix.n vmcore.n
```

If you have a multiprocessor system, you must then determine which CPU crashed. The system contains the paniccpu variable, which it sets to the number of the CPU that crashed. You determine the value of that variable by issuing the following command:

```
(dbx) print paniccpu
1
```

In this case, CPU number 1 crashed.

Once you determine which CPU crashed, you must set the context for dbx by setting the dbx variable $pid. The following example shows how to set the context for dbx:

```
(dbx) set $pid = cpudata[1].cpu_proc.p_pid
```

Because CPU number 1 crashed, you specify the number 1 as the array index in the `cpudata[1].cpu_proc.p_pid` variable name. If a CPU other than number 1 crashed, replace 1 with the appropriate number.

Once you have set the context for `dbx` or if you have only one CPU, you can use the `dbx` command `where` to display a stack trace, as shown:

```
(dbx) where
```

The following list describes `dbx` commands that are useful in kernel debugging:

- *address / count mode*

  Display the contents of the specified address. Replace `count` with the number of locations you want to display and `mode` with one of the following modes:

  – d or D, which spcecify short or longword decimal notation

  – o or O, which specify short or longword octal notation

  – x or X, which specify short or longword hexadecimal notation

  – c, which specifies a byte as type `char`

  – s, which specifies null-terminated string

  – f, which specifies single-precision real format

  – g, which specifies double-precision real format

  – i, which specifies machine instructions

  For example, if the system reported an EPC of 0x8000dead when it crashed, you can use `dbx` to determine where in the kernel that PC is located. The following command decodes nine instructions (and shows line numbers) starting at 0x8000dead. Note that code that is conditioned out (with #ifdef statements) does not count in `dbx`'s line numbering.

  ```
  (dbx) 0x8000dead/9i
  8000dead  bleq  8000deaf
  8000deaf  cvtfd *-18074(r0),$0.5
  8000deb4  movl  (r9),(r6)
  8000deb7  decl  8015fe28
  8000debd  movl  r7,r1
  8000dec0  mfpr  $12,r0
  8000dec3  mtpr  r1,$12
  8000dec6  ret
  8000dec7  halt
  (dbx)
  ```

- `print` *gnode* [*n*]

  Display the gnode structure *n* in the gnode table

- `print` *text* [*n*]

  Display the text structure *n* in the text table

- `set $pid=` *n*

  Set process context (current process) to process ID *n* (Allows you to issue `trace`, `print *up`, `print *up.u_procp`, and so on for that process)

- `print *up`

  Display the u_area of the current process

- `print *up.u_procp`

  Display the process structure of the current process ID

To debug a running kernel, issue the following command:

```
% dbx -k /vmunix
```

Then use `dbx` commands to examine the kernel. You might also find the following two commands useful when you debug a running kernel:

- `&` *`<symbol>`* `/` *`<mode>`*

  Display the address and contents of the specified symbol. The `dbx` debugger displays the contents in the specified mode. For a list of modes, see the *`address / count mode`* command in the preceding list.

- `assign` *`symbol`* `=` *`value`*

  Assign the *`value`* to the named symbol. (You must be logged in as root to change the value of symbols in a running kernel.)

D.1.3 Getting a Stack Trace on Any Process

To perform a stack trace on a process, get the PID of the process to be traced by issuing the `ps` command:

```
% ps -klax vmunix.n vmcore.n
```

The `ps` options have the following meanings:

- The `-k` option specifies using the kernel file `vmcore.n` instead of `/dev/kmem` and `/dev/mem`.
- The `-l` option displays the process status in long format, giving more information than the default display.
- The `-a` option displays all processes (not just your own) associated with a terminal.
- The `-x` option shows processes not associated with a terminal.

See `ps`(1) in the *ULTRIX Reference Pages* for complete information.

Invoke `dbx` and set `$pid` to the PID of the process you wish to examine. For example:

```
(dbx) set $pid = 1125
```

Now you can execute `trace`, `print *up`, `print *up.u_procp`, and other commands on process 1125.

Any process stored registers in the u_area is in exception frame format, and you can display the registers by issuing the following `dbx` command:

```
(dbx) print up.u_ar0
```

D.1.4 Examining the Exception Frame

All error traps and interrupts (except cache parity errors) generate an exception condition. Exception conditions trap to VECTOR(exception) in `locore.s`. The exception routine saves state in the exception frame (on the stack).

For interrupts, VECTOR(VEC_int) is called, which saves additional state on the exception frame, and calls `intr()` (in `trap.c`). The `intr()` routine calls the specific interrupt handler through c0vec_tbl.

For traps, the individual trap routines are called through the causevec. These routines [`VEC_addrerr()`, `VEC_ibe()`, and `VEC_dbe()`] in turn call `VECTOR(VEC_trap)`, which saves additional state on the exception frame, and calls `trap()` (in `trap.c`).

A pointer to the exception frame (EP) is passed as an argument to the following routines: `trap()`, `intr()`, `tlbmod()`, `tlbmiss()`, and `syscall()`. Therefore, by using `dbx` to get a trace, you can determine the address of the exception frame by displaying the EP argument. You can then display the exception frame with a `dbx` command such as:

```
(dbx) 0xffffnnnn/41X
```

The offsets within the exception frame are defined as follows (see `/sys/machine/mips/reg.h`):

```
#define EF_ARGSAVE0    0   /* arg save for c calling seq */
#define EF_ARGSAVE1    1   /* arg save for c calling seq */
#define EF_ARGSAVE2    2   /* arg save for c calling seq */
#define EF_ARGSAVE3    3   /* arg save for c calling seq */
#define EF_AT          4   /* r1:  assembler temporary */
#define EF_V0          5   /* r2:  return value 0 */
#define EF_V1          6   /* r3:  return value 1 */
#define EF_A0          7   /* r4:  argument 0 */
#define EF_A1          8   /* r5:  argument 1 */
#define EF_A2          9   /* r6:  argument 2 */
#define EF_A3         10   /* r7:  argument 3 */
#define EF_T0         11   /* r8:  caller saved 0 */
#define EF_T1         12   /* r9:  caller saved 1 */
#define EF_T2         13   /* r10: caller saved 2 */
#define EF_T3         14   /* r11: caller saved 3 */
#define EF_T4         15   /* r12: caller saved 4 */
#define EF_T5         16   /* r13: caller saved 5 */
#define EF_T6         17   /* r14: caller saved 6 */
#define EF_T7         18   /* r15: caller saved 7 */
#define EF_S0         19   /* r16: callee saved 0 */
#define EF_S1         20   /* r17: callee saved 1 */
#define EF_S2         21   /* r18: callee saved 2 */
#define EF_S3         22   /* r19: callee saved 3 */
#define EF_S4         23   /* r20: callee saved 4 */
#define EF_S5         24   /* r21: callee saved 5 */
#define EF_S6         25   /* r22: callee saved 6 */
#define EF_S7         26   /* r23: callee saved 7 */
#define EF_T8         27   /* r24: code generator 0 */
#define EF_T9         28   /* r25: code generator 1 */
```

```
#define EF_K0          29   /* r26: kernel temporary 0 */
#define EF_K1          30   /* r27: kernel temporary 1 */
#define EF_GP          31   /* r28: global pointer */
#define EF_SP          32   /* r29: stack pointer */
#define EF_S8          33   /* r30: callee saved 8 */
#define EF_RA          34   /* r31: return address */
#define EF_SR          35   /* status register */
#define EF_MDLO        36   /* low mult result */
#define EF_MDHI        37   /* high mult result */
#define EF_BADVADDR    38   /* bad virtual address */
#define EF_CAUSE       39   /* cause register */
#define EF_EPC         40   /* program counter */
```

D.1.5 Debugging Hung Systems

When your system is hung, it might display the Program Counter (PC) and Stack Pointer (SP) on the console when it crashes. In this case, you can get a stack trace easily using `dbx` and then examine the stack frame to determine what routine was executing when the system crashed. Section D.1.5.1 describes using `dbx` to get a stack trace when you have the PC and SP. Section D.1.5.3 describes examining the stack frame.

If your system does not display the PC and SP on the console when it crashes, you must get that informatiom from a core dump. However, when you are debugging a hung system, the values saved in the u_area for the currently active process are the old values that the system saved the last time it moved the process out of memory for a context switch. In this case, you must find the real kernel stack as described in Section D.1.5.2. Then, you can examine the stack frame as described in Section D.1.5.3.

D.1.5.1 Using dbx to Perform a Stack Trace

Follow these steps to perform a stack trace using `dbx` on a hung system:

1. Determine whether the idle process is running by issuing the command in the following example. If a CPU other than number 1 crashed, replace 1 with the appropriate CPU number. If you have only one CPU, replace 1 with 0.

   ```
   (dbx) &cpudata[1].cpu_noproc/d
   1
   ```

   If `dbx` displays a 1, the idle process is running.

2. Set the `$pid` `dbx` variable. The value you assign to the variable depends upon whether the idle process is running.

   Set the context for `dbx` to the idle process by issuing the following command:

   ```
   (dbx) set $pid = 3
   ```

   However, if the idle process is not running, set the context for `dbx` as follows:

   ```
   (dbx) set $pid = cpudata[1].cpu_proc.p_pid
   ```

   If a CPU other than number 1 hung, replace 1 with the appropriate CPU number. If you have only one CPU, replace 1 with 0.

3. Set the `$pc` and `$sp` variables.

Set the `$pc` variable to the PC address displayed on the console. Likewise, set the `$sp` variable to the SP address displayed on the console. The following example shows setting these two variables:

```
(dbx) assign $pc = 0x80040f20
0x80040f20
(dbx) assign $sp = 0xffffd290
0xffffd290
```

4. Perform the stack trace by issuing the `where` command as follows:

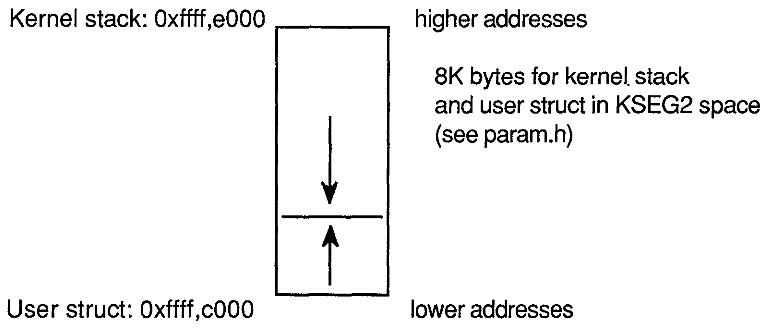
```
(dbx) where
```

D.1.5.2 Finding the Real Kernel Stack

If your system does not display a PC or SP value, you must determine where the real kernel stack is.

The kernel stack for each process in the system is located at virtual address 0xffffe000 in KSEG2 space. The system has an array of NPROC u_areas that are 8K bytes each. Each u_area contains the user struct and kernel stack for the process. Even though each user process has its u_area at the same virtual address in KSEG2 space, each u_area is mapped to a unique physical address. When the context switches, the first two entries in the TLB (safe entries) are established for mapping the u_area for that user process. Figure D-1 shows how the kernel stack and user structure appear in memory.

Figure D-1: Kernel Stack and User Structure in Memory



Kernel stack: 0xffff,e000
higher addresses
8K bytes for kernel stack
and user struct in KSEG2 space
(see param.h)
User struct: 0xffff,c000
lower addresses

ZK-0192U-R

Within `dbx`, you can display the kernel stack with a command such as the following:

```
(dbx) 0xffffd000/1028X
```

This command dumps the kernel stack from low to high memory (most recent events to oldest events).

D.1.5.3 Examining Stack Frames

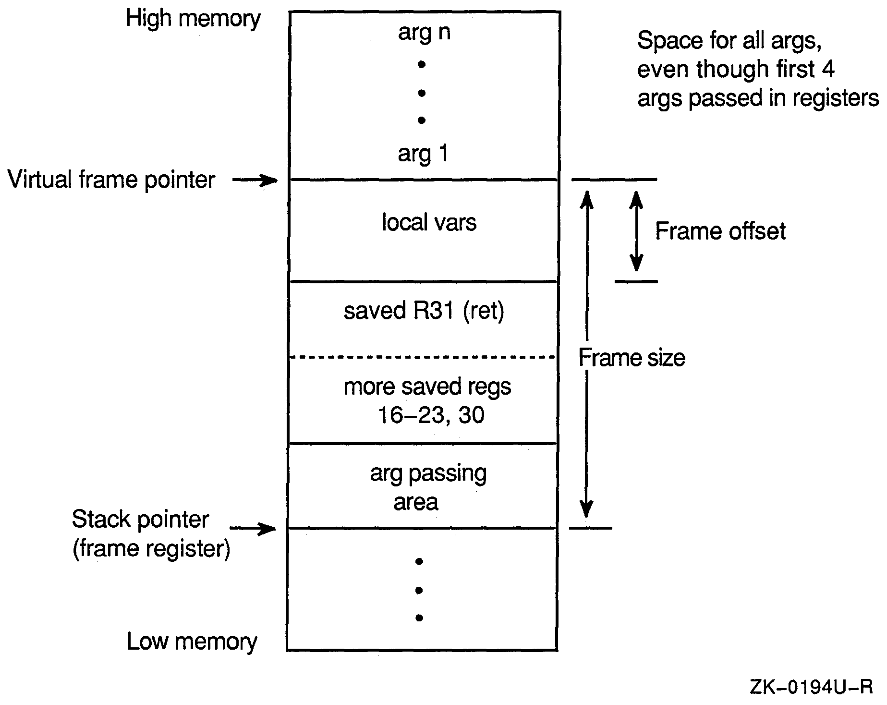
Use the odump utility to create a symbol table dump of vmunix.n:

```
% odump -P /vmunix.n > vmunix.syms
```

(See the runtime_pdr structure in the file /usr/include/sym.h for the format of the run-time procedure descriptor created by the loader.)

The fpoff field as shown by odump is the frame size for the particular procedure entry. Figure D-2 illustrates the general format of the stack (stack frames).

Figure D-2: Stack Frame in Memory



Using the symbol table dump, you can work your way back up the call history on the stack. Examples of usage are in libexc: unwind.c, exception.c, and exception.h.

It might be equally productive to start at the top of the kernel stack (high memory) and look for the return address of VEC_syscall( ) on the stack. This return address is where VEC_syscall( ) calls syscall( ), and where the stack frame for entry into syscall( ) has the return address of VEC_syscall( ) saved on the stack.

The following dbx command shows the instructions in VEC_syscall( ), in particular where syscall( ) was called, allowing you to see the return address on the stack:

```
(dbx) VEC_syscall/30i
[VEC_syscall, 0x800c3868]          ori      r5,r16,0x1
[VEC_syscall:590, 0x800c386c]      mtc0     r5,sr
[VEC_syscall:591, 0x800c3870]      sw       r2,20(sp)
[VEC_syscall:592, 0x800c3874]      sw       r3,24(sp)
[VEC_syscall:593, 0x800c3878]      move     r5,r2
[VEC_syscall:594, 0x800c387c]      move     r6,r16
```

```
[VEC_syscall:595, 0x800c3880]        jal     syscall
[VEC_syscall:595, 0x800c3884]        nop
[VEC_syscall:596, 0x800c3888]        bne     r2,r0,0x800c3810
[VEC_syscall:596, 0x800c388c]        nop
```

The return address is 0x800c3888. Using `dbx` and the dump of the kernel stack, you can examine the stack to determine what happened to the system.

D.1.6 Forcing a Panic on a System That Is Not Hung

To force a panic on a system that is not hung, login as `root` and issue the following command:

```
# dbx -k /vmunix /dev/mem
```

The following `dbx` command forces a panic on the next network interrupt, even in single-user mode (do not issue this command on diskless systems because it will not dump):

```
(dbx) assign ln_softc=0
```

The following command also panics the system:

```
(dbx) assign gnodeops=0
```

Note

Do not overwrite the process structure, because `dbx` will not be able to work on the image. Do not overwrite the console structures, because you will not see the panic messages.

D.1.7 Forcing a Memory Dump on a DS2100 or DS3100

To force a memory dump on a DS2100 or DS3100 system, press the restart button. Pressing the restart button halts the machine and clears memory, unless the `bootmode` variable is first set to `r` (restart). The following example shows how to set the `bootmode` variable (>>> represents the console prompt):

```
>>> setenv bootmode r
```

With the `bootmode` variable set to `r`, pressing the restart button dumps memory and reboots the machine. The dump might be silent and take several minutes.

D.1.8 Forcing a Memory Dump on a DS5000

To force a memory dump on a DS5000, press the restart button. Pressing the restart button halts the machine and clears memory, unless the `haltaction` variable is first set to `r` (restart). The following example shows how to set the `haltaction` variable (>>> represents the console prompt):

```
>>> setenv haltaction r
```

With the `haltaction` variable set to `r`, pressing the restart button dumps memory and reboots the machine. The dump might be silent and take several minutes.

D.1.9 Forcing a Memory Dump on a DS5400 or DS5800

To force a memory dump on a DS5400 or DS5800 system, follow these steps:

1. Set the break enable switch to the up position (pointing to the dot in the circle).
2. Press the break key to get the console prompt (>>>).
3. Run the memory dump routine by issuing the `go` command with the kernel start address + 8. For example, suppose the kernel start address is 0x80030000; in this case, the command is as follows:

```
>>> go 0x80030008
```

D.1.10 Console Commands

The following list gives the syntax for a number of console commands that are useful for debugging a RISC kernel:

- The following command dumps the contents of memory starting at the specified address and displays the specified number of longwords in hexadecimal format:

  dump -w -x *address#count*

  The following command dumps the contents of memory, starting at *address1*, ending at *address2*. The output is displayed as longwords in hexadecimal format:

  dump -w -x*address1***:***address2*

  The following command dumps the startup stack:

  dump -w -x 0x8001f400:0x8001f800

- The following command examines a byte, halfword, or word at the specified virtual address (To examine physical location 0, use address 0x8000 0000.):

  e [**-b** | **-h** | **-w**] *address*

- The following command transfers control to given entry point:

  go [*pc*]

- The following commands display help information for the specified command or, if no *command* is given, the command menu:

  help [*command*]

  ? [*command*]

- The following command displays the current value of the specified environment variable, or if no variable is specified, all environment variables:

  printenv [*var*]

- The following command sets the specified environment variable to the specified string:

  setenv *var string*

- The following command deletes the specified environment variable:

  unsetenv *var*

- The following command tests all components and subsystems:

  t a

- The following command constrains memory size to the specified number of bytes:

  boot memlimit=*bytes*

D.2 VAX Kernel Debugging

This section shows how to debug the ULTRIX kernel, `/vmunix`, on a VAX system. In addition to the information in this section, you might find *A Tutorial Introduction to ADB* in the *Supplementary Documents, Volume 2: Programmer* useful when you debug `/vmunix`. Refer also to the following header files:

- `/sys/h/proc.h`
- `/sys/h/user.h`
- `/sys/VAX/pcb.h`
- `/sys/VAX/trap.h`

The `crash` System V program might also be useful

Normally, you debug `/vmunix`, when your system crashes. On a VAX processor, crashes typically occur because of a hardware trap, hardware machine check, or software panic. The following list describes these types of crashes and the actions the system takes during the crash:

- Hardware trap

  When a hardware trap occurs, the system pushes the PSL, PC, code, and trap type onto the interrupt stack. Depending on the trap type, the code is often the last virtual address that was accessed, and is therefore the code that caused the trap (see `/sys/VAX/trap.h` for an explanation of trap types). The ULTRIX trap routine, `/sys/VAX/trap.c`, is called through the SCB. The trap routine, in turn, calls the panic routine.

  An example of a trap is a process that accesses an address outside the process's address space, which causes trap type 8, a segmentation fault.

- Hardware machine check

  When a hardware machine check occurs, the system pushes a processor dependent machine check frame onto the interrupt stack. The ULTRIX machine check routine, `/sys/vax/machdep.c`, is called through the SCB. If unrecoverable, the machine check calls the panic routine.

An example of a machine check is a parity memory error.

- Software panic

  When a software panic occurs, the kernel software detects an internal inconsistency while the system is running on the kernel stack. The kernel routine that detects the inconsistency calls the panic routine (see the *VAX Architecture Handbook* for more information).

D.2.1 Using nm to Determine Where a Crash Occurred

For a system crash that gives a PC on the console, you can use `nm` to determine which routine was executing. The following command displays the name list (symbol table) of the vmunix image in numerical order:

```
% nm -n /vmunix
```

To determine which routine was executing, find the address that is closest to, but less than, the PC from the crash; this address is the starting address of the routine executing when the system crashed. Subtract the starting address of this routine from the faulting PC to get the offset from the beginning of the routine in which the error occurred. Then, use the `adb` debugger to find the incorrect instruction.

D.2.2 Forcing a Crash Dump

If the system is hung, you can force a crash dump. To do this, halt the processor and enter console mode. (In this section, >>> represents the console prompt.) Then, issue the following command to get the address of the crash dump routine:

```
>>> E/P/L 4                    ! Get address of dump routine
  P 00000004  00001C00
```

The console's response is the address of the crash dump routine, which can then be run by typing:

```
>>> D PSL 041F0000          ! Set PSL to interrupt stack and IPL to 31
>>> S 80001C00              ! Run the dump routine
```

If the interrupt stack is invalid, the crash dump routine is not called. (The interrupt stack is in kernel address space, starting just below the address of the crash dump routine [`doadump`], and growing down in memory. The interrupt stack has a fixed size of several pages.)

There is another way to force a crash dump. But first, examine the PC and stack pointers, noting their values, because they will be changed by the commands to force a dump:

```
>>> E/G F                   ! Examine general register F (PC)
  G 0000000F  80001EAD
>>> E PSL                   ! Examine the PSL
  M 00000000  04C10004
>>> E SP                    ! Examine the stack pointer
  G 0000000E  000393E8
>>> E/I 0                   ! Examine internal register 0 (KSP)
  I 00000000  7FFFFDAC
>>> E/I 3                   ! Examine internal register 3 (USP)
  I 00000003  7FFFE2F4
>>> E/I 4                   ! Examine internal register 4 (ISP)
  I 00000004  80000C00
```

Now set the PC to -1, and continue:

```
>>> D/G F FFFFFFFF          ! Deposit -1 in PC
>>> D PSL 001F0000          ! Set IPL at 31 to block interrupts
>>> C                       ! Continue processing
```

The preceding commands force a segmentation fault, causing a crash dump. Unfortunately, some machine state is changed using this method. However, all disk writes are completed (as if `sync` had been executed).

If neither of the prior methods work, you might still be able to get a crash dump by initializing the processor before starting the crash dump routine. Initializing the processor sets it to a known state, which includes setting the PSL to run on the interrupt stack, setting the IPL to 31, and disabling memory mapping. Unfortunately, even more machine state is changed; depending on the processor, the initialization might corrupt the ISP, KSP, P0BR, P0LR, P1BR, and P1LR.

```
>>> E/P/L 4                 ! Get address of dump routine
  P 00000004  00001C00
>>> I                       ! Initialize the processor
>>> S 80001C00              ! Run dump routine
```

D.2.3 Getting a Stack Trace on Any Process

To perform a stack trace on a process, get the PID of the process to be traced by issuing the `ps` command:

```
% ps -klax vmunix.n vmcore.n
```

The `ps` options have the following meanings:

-k	Use kernel file (`vmcore.n` instead of `/dev/kmem` and `/dev/mem`)
-l	Display in long format, giving more information
-a	Show all processes (not just your own) associated with a terminal
-x	Show processes not associated with a terminal

See `ps`(1) in the *ULTRIX Reference Pages* for complete information.

The preceding `ps` command displays the PIDs of every process on the system. Note the PID of the process you are interested in. Now issue the `/etc/pstat` command with the -p, -a, and -k options:

```
% pstat -pak  vmunix.n vmcore.n
```

The `pstat` options have the following meanings:

-p	Display process table for active processes
-a	Describe all process slots
-k	Required option when a core file is specified

See `pstat`(8) in the *ULTRIX Reference Pages* for complete information.

The following shows an example of `pstat` output:

```
195/1044 processes
   LOC     S     F POIP PRI       SIG  UID SLP TIM  CPU  NI    PGRP     PID    PPID
  ADDR    RSS SRSS SIZE    WCHAN     LINK    TEXTP CLKT  TTYP
801e5f70   1     3    0   0          0    0    0 127    0  20       0       0       0
   c96     0     0     0   15f936  1ea170       0          0
801e6030   1     1    0  30          0    0   87 127    0  20       0       1       0
  96df   1c6     0   1f0    1e6030  1e9f30  218e80          0
```

```
801e60f0  1    3    0   1         0    0 127 127    0  20       0       2       0
  96bf    0    0  2000   1e60f0        0        0          0
```

Locate the PID you want in the PID field, second from right. The process's location (the memory location of the process structure) is in the LOC field, the leftmost field. (In the preceding example, the location of process 0 is 801e5f70.) Check the process's state and flag codes, second and third fields, labeled S and F.

Invoke adb with the following command:

```
% adb -k vmunix.n vmcore.n
```

The following adb commands yield the address of the u_area of process 0 (from the preceding example):

```
801e5f70/X     ! Show contents of process structure's first field
801e5f70: 8000feff
    Return    ! Show contents of process structure's second field
801e5f74: 80f03a00
     Return    ! Show contents of process structure's third field
801e5f7c: 81000fff
     Return    ! Show contents of process structure's fourth field
801e5f80: 801ee3e0
     Return    ! Show contents of process structure's fifth field
801e5f88: 80a20fff
```

The fifth field in the process structure contains the address that maps the u_area (see proc.h: proc struct and p_addr field); the following adb commands set a stack trace for the process:

```
80a20fff$p      ! Set process context for adb
$c              ! Trace stack of process in question
```

D.2.4 adb Command Summary

You can invoke the adb debugger to debug either a crash image or a running system. To invoke adb on a crash image, issue the following command:

```
% adb -k vmunix vmcore
```

To invoke adb on a running system, issue the following command:

```
% adb -k -w /vmunix /dev/mem
```

Once you invoke adb you can use a number of commands to perform debugging tasks. Several adb commands allow you to specify a format for the output from the command. The following list describes the format characters you use on the command line to control the format of adb's output:

- d – specifies signed decimal word output
- D – specifies signed decimal longword output
- f – specifies floating point longword output
- F – specifies floating point double output
- o – specifies unsigned octal word output
- O – specifies unsigned octal longword output

- `q` – specifies signed octal word output
- `Q` – specifies signed octal longword output
- `s` – specifies string output
- `u` – specifies unsigned decimal word output
- `U` – specifies unsigned decimal longword output
- `x` – specifies hexadecimal word output
- `X` – specifies hexadecimal longword output

The following list describes several commonly used `adb` commands:

- `?` [*address*] *f*

  Display, using the format character *f*, values in the disk image starting at *address*.

- / [*address*] *f*

  Display, using the format character *f*, values in the core file starting at *address*.

- `=` *f*

  Display, using the format character *f*, the virtual address of a symbol.

- `*(scb-4)$c`

  Trace stack of whichever stack was currently active (interrupt or kernel) in this format:

  ```
  func_3 (args) from addr_3       (newest)
  func_2 (args) from addr_2
  func_1 (args) from addr_1       (oldest)
  ```

  The `func_1()` routine calls `func_2()` from addr_1 in `func_1()`. Therefore, the stack frame with the saved PC of addr_1 (return address), is the stack frame of `func_2()`.

- *address* `$c`

  Trace stack starting from *address*

- *routine-name* +2 [/] [?] `i`

  Display assembly instructions starting at the beginning of the named routine (*+2* skips over the register save mask)

- *address* [/] [?] `i`

  Display assembly instructions starting at *address*

- `Return`

  Examine the location after the last examined location

- `^@`

  Examine the location before the last examined location

- [/] [?] `w` *value*

  Write *value* to the last addressed location

- `$R`

  Show register contents
- *`range`* `$S`

  Extend range of symbolic names

D.2.5 adb Scripts

The directory `/usr/lib/adb` contains `adb` scripts that format kernel data structures. The following list shows some ways you might use the scripts:

- *`address`* `$<` *`script`*

  Apply *`script`* at *`address`*
- *`u_block`* `$<u`

  Apply the user structure script at symbolic address *`u_block`* (the current u block; that is, the user structure of the current process)
- *`address`* `$<proc`

  Apply the proc script at *`address`*, obtained from the user structure
- *`address`* `$<pcb`

  Apply the pcb script at *`address`*

D.2.6 Examining Stack Frames with adb

Using `adb` to examine stack frames is useful for seeing values of local variables. The following `adb` commands are useful in examining stack frames:

- `(scb-4)/X`

  Display the address of the current stack, which is stored in `scb-4`. If the address is 800*nnnnn*, the system was using the interrupt stack when it crashed; if the address is 7ff*nnnnn*, the system was using the kernel stack.
- *`intstack`*`/20X`

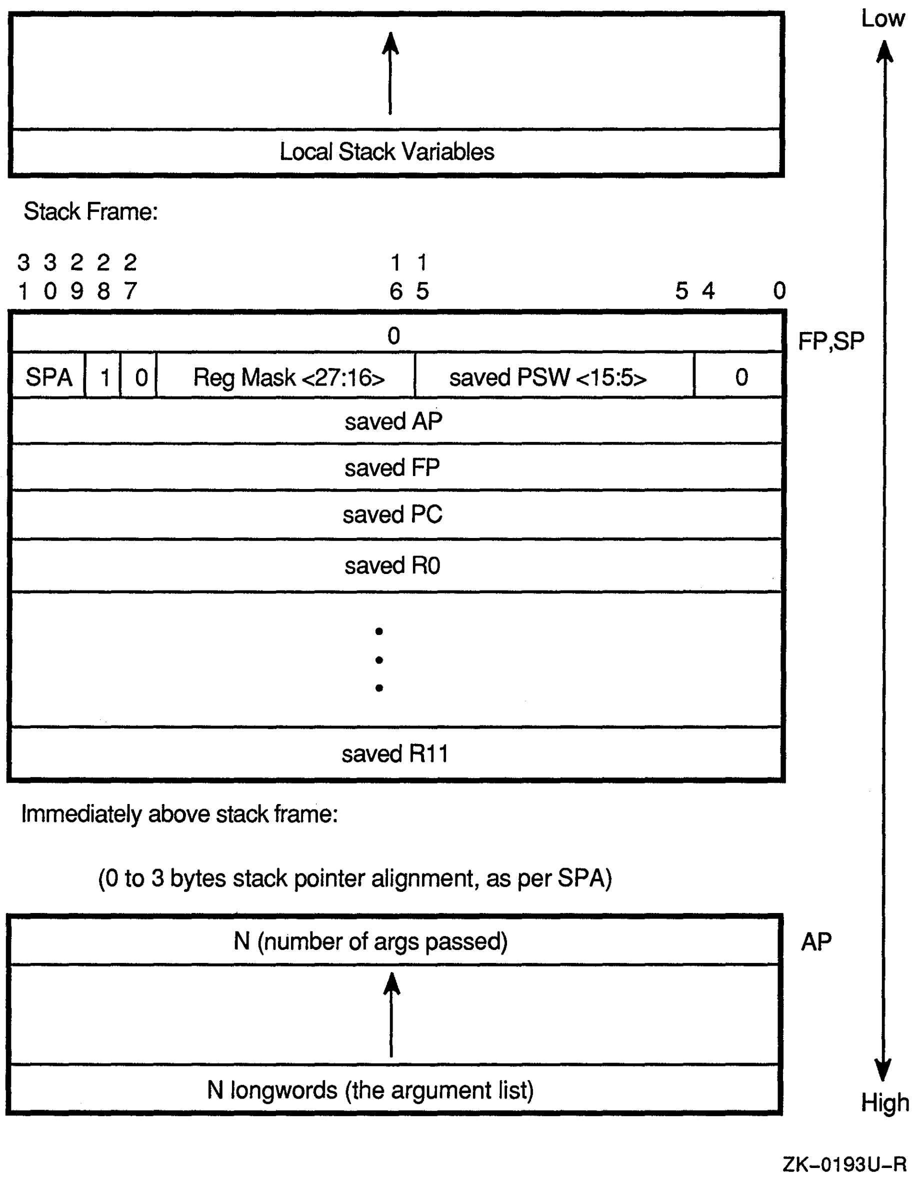
  Starting at the address of *`intstack`*, display 20 longwords in hexadecimal format.
- `u$<u`

  Show the first item in the user structure, which is the kernel stack pointer (KSP).
- *`KSP`*`/20X`

  Starting at the address of KSP, display 20 longwords in hexadecimal format.

To find a stack frame (a call frame for a procedure call), look for a 0 longword (condition handler) followed by a longword with bit 29 set, which indicates a call (for example, 2e000000). Figure D-3 illustrates a stack frame in memory.

Figure D-3: Stack Frame in Memory



The `calls` instruction pushes the argument count onto the stack, then aligns the stack and creates the stack frame (call frame), which is the saved register through the condition handler. (For more information, see the *VAX-11 Architecture Reference Manual.*)

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	**Call**	**Contact**
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to Languages and Programming
AA-ML94C-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

What do you like best about this manual? ______________________________

What do you like least about this manual? ______________________________

Please list errors you have found in this manual:

Page	Description

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? __________

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________

______________ Email ______________ Phone __________

------ Do Not Tear – Fold Here and Tape --

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–2698

------ Do Not Tear – Fold Here --

Cut
Along
Dotted
Line