

Dansk Data Elektronik A/S

**SUPERMAX
Programmer's Guide
System V Release 3.1
Version 2.0**

©1986 AT&T, USA

**©1989 Dansk Data Elektronik A/S, Denmark
Version 1.0, First Edition, published December 1989**

**©1991 Dansk Data Elektronik A/S, Denmark
Version 2.0, revised and published January 1991**

**©1993 Dansk Data Elektronik A/S, Denmark
Version 2.0, Chapter 18 added and published March 1993**

**All Rights Reserved
Printed in Denmark**

Stock no.: 94300441

NOTICE

The information in this document is subject to change without notice. AT&T or Dansk Data Elektronik A/S, Denmark assumes no responsibility for any errors that may appear in this document.

UNIX is a registered trademark of AT&T in the USA and other countries.

SUPERMAX is a registered trademark of Dansk Data Elektronik A/S, Denmark.

PDP is a trademark of Digital Equipment.

Teletype is a registered trademark of AT&T.

VT100 is a trademark of Digital Equipment.

Table of Contents

	Page
List of Figures	ix
Purpose	xv

PART 1: PROGRAMMING

Chapter 1: Programming in a UNIX System Environment: An Overview

Introduction	1- 1
UNIX System Tools and Where You Can Read About Them	1- 5
Three Programming Environments	1- 8
Summary	1-10

Chapter 2: Programming Basics

Introduction	2- 1
Choosing a Programming Language	2- 3
Special Purpose Language	2- 7
After Your Code Is Written	2-11
The Interface Between a Programming Language and the UNIX System	2-17
Analysis/Debugging	2-53
Program Organizing Utilities	2-73

Table of Contents

	Page
Chapter 3: Application Programming	
Introduction	3 - 1
Application Programming	3 - 3
Language Selection	3 - 7
Advanced Programming Tools	3 - 17
Programming Support Tools	3 - 27
Project Control Tools	3 - 39
liber, A Library System	3 - 43
 PART 2: SUPPORT TOOLS	
Chapter 4: awk	
Introduction	4 - 1
Basic awk	4 - 3
Patterns	4 - 15
Actions	4 - 23
Output	4 - 43
Input	4 - 49
Using awk with Other Commands and the Shell	4 - 57
Example Applications	4 - 61
awk Summary	4 - 67
 Chapter 5: lex	
An Overview of lex Programming	5 - 1
Writing lex Programs	5 - 3
Running lex under the UNIX System	5 - 21

	Page
Chapter 6: yacc	
Introduction.....	6- 1
Basic Specifications.....	6- 5
Parser Operation.....	6-15
Ambiguity and Conflicts.....	6-21
Precedence.....	6-27
Error Handling.....	6-31
The yacc Environment.....	6-35
Hints for Preparing Specifications.....	6-37
Advanced Topics.....	6-43
Examples.....	6-51
Chapter 7: File and Record Locking	
Introduction.....	7- 1
Terminology.....	7- 3
File Protection.....	7- 5
Selecting Advisory or Mandatory Locking.....	7-18
Chapter 8: Interprocess Communication	
Introduction.....	8- 1
Messages.....	8- 3
Getting Message Queues.....	8- 8
Controlling Message Queues.....	8-17
Operations for Messages.....	8-24
Semaphores.....	8-39
Shared Memory.....	8-77
Operations for Shared Memory.....	8-103

Table of Contents

	Page
Chapter 9: curses/terminfo	
Introduction.....	9 - 1
Overview	9 - 3
Working with curses Routines	9 - 9
Getting Simple Output and Input.....	9 - 20
Controlling Output and Input	9 - 41
Building Windows and Pads	9 - 53
New Windows	9 - 59
Using Advanced curses Features	9 - 63
Working with terminfo Routines.....	9 - 69
Working with the terminfo Database	9 - 77
curses Program Examples	9 - 91
Chapter 10: The Common Object File Format (COFF)	
Introduction.....	10 - 1
Definitions and Conventions.....	10 - 3
Optional Header Information	10 - 7
Section Headers.....	10 - 10
Sections	10 - 13
Relocation Information.....	10 - 13
Line Numbers.....	10 - 16
Symbol Table.....	10 - 18
Storage Classes.....	10 - 26
Type Entry.....	10 - 32
Auxillary Table Entries	10 - 38
String Table.....	10 - 47

	Page
Chapter 11: The Link Editor	
The Link Editor	11- 1
Link Editor Command Language	11- 5
Notes and Special Considerations	11-25
Syntax Diagram for Input Directives.....	11-37
Chapter 12: make	
Introduction	12- 1
Basic Features	12- 2
Description Files and Substitutions	12- 9
Recursive Makefiles.....	12-15
Source Code Control System Filenames: the Tilde	12-21
Command Usage.....	12-27
Suggestions and Warnings	12-31
Internal Rules.....	12-33
Chapter 13: Source Code Control System (SCCS)	
Introduction	13- 1
SCCS For Beginners	13- 3
Delta Numbering.....	13- 9
SCCS Command Conventions.....	13-13
SCCS Commands.....	13-15
SCCS Files	13-45

	Page
Chapter 14: SDB/DBX – The Symbolic Debugger	
Preface.....	14 – 1
sdb	
Introduction to sdb	14 – 3
Using sdb	14 – 5
Source File Display and Manipulation.....	14 – 10
A Controlled Environment for Program Testing.....	14 – 12
Machine Language Debugging.....	14 – 16
Other Commands	14 – 17
dbx	
Debugging Your Code with dbx	14 – 19
Introduction to dbx	14 – 23
Running DBX	14 – 27
Using DBX Commands.....	14 – 31
Working with the DBX Monitor.....	14 – 39
Controlling DBX.....	14 – 45
Examining Source Programs with DBX	14 – 67
Controlling Your Program with DBX	14 – 77
Setting Breakpoints with DBX.....	14 – 87
Examining Program State with DBX.....	14 – 99
Debugging at the Machine Level with DBX.....	14 – 109
DBX Debugger Command Summary.....	14 – 119
DBX Sample Program.....	14 – 127

	Page
Chapter 15: lint	
Introduction.....	15 - 1
Usage.....	15 - 3
lint Message Types.....	15 - 5
Chapter 16: C Language	
Introduction.....	16 - 1
Lexical Conventions.....	16 - 3
Storage Class and Type.....	16 - 9
Operator Conversions.....	16 - 13
Expressions and Operators.....	16 - 17
Declarations.....	16 - 31
Statements.....	16 - 49
External Definitions.....	16 - 57
Scope Rules.....	16 - 61
Compiler Control Lines.....	16 - 65
Types Revisited.....	16 - 71
Constant Expressions.....	16 - 77
Portability Considerations.....	16 - 79
Syntax Summary.....	16 - 81
Chapter 17: Improving Program Performance	
Introduction.....	17 - 1
Optimization.....	17 - 3
Overview.....	17 - 3
Optimization Options.....	17 - 7
Improving Global Optimization.....	17 - 13

Table of Contents

	Page
Improving Other Optimization.....	17 – 21
Limiting the Size of Global Pointer Data.....	17 – 23
Chapter 18: Native Language Support	
Introduction.....	18 – 1
The NLS Database	18 – 1
Configuration Data.....	18 – 1
Collating Sequence Tables.....	18 – 2
Character Classification Tables.....	18 – 3
Shift Tables	18 – 3
Language Information	18 – 3
Program Localisation.....	18 – 5
The Program Locale.....	18 – 5
The Announcement Mechanism	18 – 6
Announcement Categories.....	18 – 8
Setting the Program Locale	18 – 9
Setting a Specific Category to the Implementation-defined Default.....	18 – 10
Setting all Categories to the Implementation-defined Default	18 – 10
Message Catalogues	18 – 13
Introduction	18 – 13
Creating a Message Catalogue	18 – 13
Message Text Source Files	18 – 14
Gencat.....	18 – 16
Accessing a Message Catalogue.....	18 – 17
Catgets.....	18 – 17

Table of Contents

	Page
Search and Naming Conventions.....	18 - 19
C Program Locale	18 - 21
Coded Character Set	18 - 21
Cultural Data.....	18 - 27
Additional References.....	18 - 28
Glossary	G - 1
Index to Utilities	I - 1

List of Figures

	Page
Figure 2- 1: Using Command Line Arguments to Set Flags .	2-19
Figure 2- 2: Using <code>argv[n]</code> Pointers to Pass a Filename	2-20
Figure 2- 3: C Language Standard I/O Subroutines.....	2-23
Figure 2- 4: String Operations	2-25
Figure 2- 5: Classifying ASCII Character-Coded Integer Values	2-27
Figure 2- 6: Conversion Functions and Macros.....	2-28
Figure 2- 7: Manual Page for <code>gets(3S)</code>	2-30
Figure 2- 8: How <code>gets</code> Is Used in a Program	2-32
Figure 2- 9: A Version of <code>stdio.h</code>	2-33
Figure 2-10: Environment and Status System Calls	2-42
Figure 2-11: Process Status.....	2-43
Figure 2-12: Example of <code>fork</code>	2-47
Figure 2-13: Example of a <code>popen</code> pipe.....	2-49
Figure 2-14: Signal Numbers Defined in <code>/usr/include/sys/signal.h</code>	2-51
Figure 2-15: Source Code for Sample Program	2-54
Figure 2-16: <code>cflow</code> Output, No Options	2-57
Figure 2-17: <code>cflow</code> Output, Using <code>-r</code> Option	2-58
Figure 2-18: <code>cflow</code> Output, Using <code>-ix</code> Option	2-59
Figure 2-19: <code>cflow</code> Output, Using <code>-r</code> and <code>-ix</code> Options	2-60
Figure 2-20: <code>ctrace</code> Output.....	2-62
Figure 2-21: <code>cxref</code> Output, Using <code>-c</code> Option	2-65
Figure 2-22: <code>lint</code> Output.....	2-70
Figure 2-23: <code>prof</code> Output	2-74
Figure 2-24: <code>nm</code> Output, with <code>-f</code> Option	2-77

	Page
Figure 3 – 1: The <code>fcntl.h</code> Header File	3 – 20
Figure 4 – 1: <code>awk</code> Program Structure and Example	4 – 3
Figure 4 – 2: The Sample Input File <code>countries</code>	4 – 6
Figure 4 – 3: <code>awk</code> Comparison Operators	4 – 16
Figure 4 – 4: <code>awk</code> Regular Expressions	4 – 21
Figure 4 – 5: <code>awk</code> Built-in Variables	4 – 23
Figure 4 – 6: <code>awk</code> Built-in Arithmetic Functions	4 – 26
Figure 4 – 7: <code>awk</code> Built-in String Functions	4 – 27
Figure 4 – 8: <code>awk</code> <code>printf</code> Conversion Characters	4 – 45
Figure 4 – 9: <code>getline</code> Function	4 – 54
Figure 5 – 1: Creation and Use of a Lexical Analyzer with <code>lex</code>	5 – 2
Figure 8 – 1: <code>ipc_perm</code> Data Structure	8 – 6
Figure 8 – 2: Operation Permissions Codes	8 – 10
Figure 8 – 3: Control Commands (Flags)	8 – 11
Figure 8 – 4: <code>msgget()</code> System Call Example	8 – 15
Figure 8 – 5: <code>msgctl()</code> System Call Example	8 – 21
Figure 8 – 6: <code>msgop()</code> System Call Example	8 – 32
Figure 8 – 7: Operation Permissions Codes	8 – 47
Figure 8 – 8: Control Commands (Flags)	8 – 48
Figure 8 – 9: <code>semget()</code> System Call Example	8 – 52
Figure 8 – 10: <code>semctl()</code> System Call Example	8 – 63
Figure 8 – 11: <code>semop(2)</code> System Call Example	8 – 73
Figure 8 – 12: Shared Memory State Information	8 – 80
Figure 8 – 13: Operation Permissions Codes	8 – 84
Figure 8 – 14: Control Commands (Flags)	8 – 85
Figure 8 – 15: <code>shmget(2)</code> System Call Example	8 – 89

List of Figures

	Page
Figure 8 - 16: shmctl(2) System Call Example	8 - 98
Figure 8 - 17: shmop() System Call Example.....	8 - 107
Figure 9 - 1: A Simple curses Program	9 - 4
Figure 9 - 2: A Shell Script Using terminfo Routines	9 - 6
Figure 9 - 3: The Purposes of initscr() , refresh() , and endwin() in a Program.....	9 - 11
Figure 9 - 4: The Relationship between stdscr and a Terminal Screen.....	9 - 16
Figure 9 - 5: Multiple Windows and Pads Mapped to a Terminal Screen	9 - 19
Figure 9 - 6: Input Option Settings for curses Programs....	9 - 48
Figure 9 - 7: The Relationship Between a Window and a Terminal Screen	9 - 56
Figure 9 - 8: Sending a Message to Several Terminals.....	9 - 68
Figure 9 - 9: Typical Framework of a terminfo Program....	9 - 70
Figure 10 - 1: Object File Format	10 - 2
Figure 10 - 2: File Header Contents.....	10 - 5
Figure 10 - 3: File Header Flags	10 - 6
Figure 10 - 4: File Header Declaration	10 - 7
Figure 10 - 5: Optional Header Contents	10 - 8
Figure 10 - 6: UNIX System Magic Numbers	10 - 8
Figure 10 - 7: aouthdr Declaration	10 - 9
Figure 10 - 8: Section Header Contents.....	10 - 10
Figure 10 - 9: Section Header Flags.....	10 - 11
Figure 10 - 10: Section Header Declaration	10 - 12
Figure 10 - 11: Relocation Section Contents	10 - 14
Figure 10 - 12: Relocation Types.....	10 - 14
Figure 10 - 13: Relocation Entry Declaration	10 - 15

	Page
Figure 10 – 14: Line Number Grouping.....	10 – 16
Figure 10 – 15: Line Number Entry Declaration.....	10 – 17
Figure 10 – 16: COFF Symbol Table.....	10 – 18
Figure 10 – 17: Special Symbols in the Symbol Table.....	10 – 20
Figure 10 – 18: Special Symbols (.bb and .eb)	10 – 21
Figure 10 – 19: Nested blocks	10 – 22
Figure 10 – 20: Example of the Symbol Table	10 – 23
Figure 10 – 21: Symbols for Functions	10 – 23
Figure 10 – 22: Symbol Table Entry Format	10 – 24
Figure 10 – 23: Name Field.....	10 – 25
Figure 10 – 24: Storage Classes	10 – 26
Figure 10 – 25: Storage Class by Special Symbols.....	10 – 28
Figure 10 – 26: Restricted Storage Classes	10 – 28
Figure 10 – 27: Storage Class and Value	10 – 29
Figure 10 – 28: Section Number.....	10 – 30
Figure 10 – 29: Section Number and Storage Class	11 – 31
Figure 10 – 30: Fundamental Types	10 – 33
Figure 10 – 31: Derived Types	10 – 34
Figure 10 – 32: Type Entries by Storage Class.....	10 – 35
Figure 10 – 33: Symbol Table Entry Declaration	10 – 37
Figure 10 – 34: Auxiliary Symbol Table Entries.....	10 – 38
Figure 10 – 35: Format for Auxiliary Table Entries for Sections.....	10 – 39
Figure 10 – 36: Tag Names Table Entries.....	10 – 40
Figure 10 – 37: Table Entries for End of Structures	10 – 40
Figure 10 – 38: Table Entries for Functions.....	10 – 41
Figure 10 – 39: Table Entries for Arrays	10 – 42
Figure 10 – 40: End of Block and Function Entries.....	10 – 42

List of Figures

	Page
Figure 10 – 41: Format for Beginning of Block and Function..	10 – 43
Figure 10 – 42: Entries for Structures, Unions, and Enumerations	10 – 43
Figure 10 – 43: Auxiliary Symbol Table Entry	10 – 45
Figure 10 – 44: String Table	10 – 47
Figure 11 – 1 : Operator Symbols	11 – 6
Figure 11 – 2 : Syntax Diagram for Input Directives	11 – 37
Figure 12 – 1 : Summary of Default Transformation Path	12 – 17
Figure 12 – 2 : make Internal Rules	12 – 33
Figure 13 – 1 : Evolution of an SCCS File.....	13 – 9
Figure 13 – 2 : Tree Structure with Branch Deltas	13 – 10
Figure 13 – 3 : Extended Branching Concept	13 – 11
Figure 13 – 4 : Determination of New SID	13 – 24

Purpose

This guide is designed to give you information about programming in a UNIX system environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming languages by concentrating on the other elements that are part of getting programs into operation.

Audience and Prerequisite Knowledge

As the title suggests, we are addressing programmers, especially those who have not worked extensively with the UNIX system. No special level of programming involvement is assumed. We hope the book will be useful to people who write only an occasional program as well as those who work on or manage large application development projects.

Programmers in the expert class, or those engaged in developing system software, may find this guide lacks the depth of information they need. For them we recommend the *System V Reference Manual*.

Knowledge of terminal use, of a UNIX system editor, and of the UNIX system directory/file structure is assumed. If you feel shaky about your mastery of these basic tools, you might want to look over the *User's Guide* before tackling this one.

Organization

The material is organized into two parts and seventeen chapters, as follows:

- Part 1, Chapter 1 – Overview

Identifies the special features of the UNIX system that make up the programming environment: the concept of building blocks, pipes, special files, shell programming, etc. As a framework for the material that follows, three different levels of programming

Purpose

in a UNIX system are defined: single-user, applications, and systems programming.

- **Chapter 2 – Programming Basics**

Describes the most fundamental utilities needed to get programs running.

- **Chapter 3 – Application Programming**

Enlarges on many of the topics covered in the previous chapter with particular emphasis on how things change as the project grows bigger. Describes tools for keeping programming projects organized.

- **Part 2, Chapters 4 through 18 – Support Tools, Descriptions, and Tutorials**

Includes detailed information about the use of many of the UNIX system tools.

The C Connection

The UNIX system supports many programming languages, and C compilers are available on many different operating systems. Nevertheless, the relationship between the UNIX operating system and C has always been and remains very close. Most of the code in the UNIX operating system is C, and over the years many organizations using the UNIX system have come to use C for an increasing portion of their application code. Thus, while this guide is intended to be useful to you no matter what language(s) you are using, you will find that, unless there is a specific language-dependent point to be made, the examples assume you are programming in C.

Hardware/Software Dependencies

The text reflects the way things work on an SUPERMAX computer running under System V Release 3.1. A SUPERMAX may be multiprocessor and heterogeneous at the same time, as it may consist of one or more CISC (Motorola 680x0) processors and maybe one or more RISC (Mips R3000) processors.

A program may be developed to run on either processor type as compilers, linkers, . . . , are controlled by the environment variable TARGETMC that can be adjusted. Different types of executable modules may be put together in one loadmodule that may run on either processor type by use of the *mkhem*(1) feature.

TARGETMC	Processor	Compatibility
68020	68020/68030	Motorola
R3KMO	R3000	Motorola
R3KMI	R3000	MIPS

By adjusting the TARGETMC a program running on a R3000 may either be BINARY compatible with MIPS (big-endian), or it will place data the MOTOROLA way.

Notation Conventions

Whenever the text includes examples of output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout UNIX system documentation:

- Commands that you type in from your terminal are shown in **bold type**.

Purpose

- Text that is printed on your terminal by the computer is shown in constant width type. Constant width type is also used for code samples because it allows the most accurate representation of spacing. Spacing is often a matter of coding style, but is sometimes critical.
- Comments added to a display to show that part of the display has been omitted are shown in *italic* type and are indented to separate them from the text that represents computer output or input. Comments that explain the input or output are shown in the same type font as the rest of the display. Italics are also used to show substitutable values, such as, *filename*, when the format of a command is shown.
- There is an implied RETURN at the end of each command and menu response you enter. Where you may be expected to enter only a RETURN (as in the case where you are accepting a menu default), the symbol `<CR>` is used.
- In cases where you are expected to enter a control character, it is shown as, for example, `CTRL-D`. This means that you press the `d` key on your keyboard while holding down the `CTRL` key.
- The dollar sign, `$`, and pound sign, `#`, symbols are the standard default prompt signs for an ordinary user and root respectively. `$` means you are logged in as an ordinary user. `#` means you are logged in as root.
- When the `#` prompt is used in an example, it means the command illustrated may be used only by root.

Command References

When commands are mentioned in a section of the text for the first time, a reference to the manual section where the command is formally described is included in parentheses: *command*(section).

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system may produce slightly different output. Some displays depend on a particular machine configuration that may differ from yours. Changes between releases of the UNIX system software may cause small differences in what appears on your terminal.

Where complete code samples are shown, we have tried to make sure they compile and work as represented. Where code fragments are shown, we have attempted to maintain the same standards of coding accuracy for them.

This page is intentionally left blank

Chapter 1: Programming in a UNIX System Environment: An Overview

	Page
Introduction.....	1- 1
The Early Days	1- 1
UNIX System Philosophy Simply Stated	1- 3
UNIX System Tools and Where You Can Read About Them	1- 5
Tools Covered and Not Covered in this Guide	1- 5
The Shell as a Prototyping Tool	1- 6
Three Programming Environments	1- 8
Single-User Programmer	1- 8
Application Programming	1- 9
Systems Programmers.....	1-10
Summary	1-10

Table of Contents

This page is intentionally left blank

Introduction

The 1983 Turing Award of the Association for Computing Machinery was given jointly to Ken Thompson and Dennis Ritchie, the two men who first designed and developed the UNIX operating system. The award citation said, in part:

"The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the UNIX system has led a generation of software designers to new ways of thinking about programming. The genius of the UNIX system is its framework which enables programmers to stand on the work of others."

As programmers working in a UNIX system environment, why should we care what Thompson and Ritchie did? Does it have any relevance for us today?

It does because if we understand the thinking behind the system design and the atmosphere in which it flowered, it can help us become productive UNIX system programmers more quickly.

The Early Days

You may already have read about how Ken Thompson came across a DEC PDP-7 machine sitting unused in a hallway at AT&T Bell Laboratories, and how he and Dennis Ritchie and a few of their colleagues used that as the original machine for developing a new operating system that became UNIX.

The important thing to realize, however, is that what they were trying to do was fashion a pleasant computing environment for themselves. It was not, "Let's get together and build an operating system that will attract world-wide attention."

Introduction

The sequence in which elements of the system fell into place is interesting. The first piece was the file system, followed quickly by its organization into a hierarchy of directories and files. The view of everything, data stores, programs, commands, directories, even devices, as files was critical, as was the idea of a file as a one-dimensional array of bytes with no other structure implied. The cleanness and simplicity of this way of looking at files has been a major contributing factor to a computer environment that programmers and other users have found comfortable to work in.

The next element was the idea of processes, with one process being able to create another and communicate with it. This innovative way of looking at running programs as processes led easily to the practice (quintessentially UNIX) of reusing code by calling it from another process. With the addition of commands to manipulate files and an assembler to produce executable programs, the system was essentially able to function on its own.

The next major development was the acquisition of a DEC PDP-11 and the installation of the new system on it. This has been described by Ritchie as a stroke of good luck, in that the PDP-11 was to become a hugely successful machine, its success to some extent adding momentum to the acceptance of the system that began to be known by the name of UNIX.

By 1972 the innovative idea of pipes (connecting links between processes whereby the output of one becomes the input of the next) had been incorporated into the system, the operating system had been recoded in higher level languages (first B, then C), and had been dubbed with the name UNIX (coined by Brian Kernighan). By this point, the "pleasant computing environment" sought by Thompson and Ritchie was a reality; but some other things were going on that had a strong influence on the character of the product then and today.

It is worth pointing out that the UNIX system came out of an atmosphere that was totally different from that in which most commercially successful operating systems are produced. The more typical atmosphere is that described by Tracy Kidder in *The Soul of a New*

Machine. In that case, dozens of talented programmers worked at white heat, in an atmosphere of extremely tight security, against murderous deadlines. By contrast, the UNIX system could be said to have had about a ten year gestation period. From the beginning it attracted the interest of a growing number of brilliant specialists, many of whom found in the UNIX system an environment that allowed them to pursue research and development interests of their own, but who in turn contributed additions to the body of tools available for succeeding ranks of UNIX programmers.

Beginning in 1971, the system began to be used for applications within AT&T Bell Laboratories, and shortly thereafter (1974) was made available at low cost and without support to colleges and universities. These versions, called research versions and identified with Arabic numbers up through 7, occasionally grew on their own and fed back to the main system additional innovative tools. The widely-used screen editor *vi*(1), for example, was added to the UNIX system by William Joy at the University of California, Berkeley. In 1979 acceding to commercial demand, AT&T began offering supported versions (called development versions) of the UNIX system. These are identified with Roman numerals and often have interim release numbers appended. The current development version, for example, is UNIX System V Release 3.1.

Versions of the UNIX system being offered now by AT&T are coming from an environment more closely related, perhaps, to the standard software factory. Features are being added to new releases in response to the expressed needs of the market place. The essential quality of the UNIX system, however, remains as the product of the innovative thinking of its originators and the collegial atmosphere in which they worked. This quality has on occasion been referred to as the UNIX philosophy, but what is meant is the way in which sophisticated programmers have come to work with the UNIX system.

UNIX System Philosophy Simply Stated

For as long as you are writing programs on a UNIX system you should keep this motto hanging on your wall:

Build on the work of others

Unlike computer environments where each new project is like starting with a blank canvas, on a UNIX system a good percentage of any programming effort is lying there in **bins**, and **lbins**, and **/usr/bins**, not to mention **etc**, waiting to be used.

The features of the UNIX system (pipes, processes, and the file system) contribute to this reusability, as does the history of sharing and contributing that extends back to 1969. You risk missing the essential nature of the UNIX system if you don't put this to work.

UNIX System Tools and Where You Can Read About Them

The term "UNIX system tools" can stand some clarification. In the narrowest sense, it means an existing piece of software used as a component in a new task. In a broader context, the term is often used to refer to elements of the UNIX system that might also be called features, utilities, programs, filters, commands, languages, functions, and so on. It gets confusing because any of the things that might be called by one or more of these names can be, and often are, used in the narrow way as part of the solution to a programming problem.

Tools Covered and Not Covered in this Guide

The *Programmer's Guide* is about tools used in the process of creating programs in a UNIX system environment, so let's take a minute to talk about which tools we mean, which ones are not going to be covered in this book, and where you might find information about those not covered here. Actually, the subject of things not covered in this guide might be even more important to you than the things that are. We couldn't possibly cover everything you ever need to know about UNIX system tools in this one volume.

Tools not covered in this text:

- the **login** procedure,
- UNIX system editors and how to use them,
- how the file system is organized and how you move around in it,
- shell programming.

UNIX System Tools

Information about these subjects can be found in the *User's Guide* and a number of commercially available texts.

Tools covered here can be classified as follows:

- utilities for getting programs running
- utilities for organizing software development projects
- specialized languages
- debugging and analysis tools
- compiled language components that are not part of the language syntax, for example, standard libraries, systems calls, and functions.

The Shell as a Prototyping Tool

Any time you log in to a UNIX system machine you are using the shell. The shell is the interactive command interpreter that stands between you and the UNIX system kernel, but that's only part of the story. Because of its ability to start processes, direct the flow of control, field interrupts and redirect input and output it is a full-fledged programming language. Programs that use these capabilities are known as shell procedures or shell scripts.

Much innovative use of the shell involves stringing together commands to be run under the control of a shell script. The dozens and dozens of commands that can be used in this way are documented in the *System V Reference Manual*. Time spent with the *System V Reference Manual* can be rewarding. Look through it when you are trying to find a command with just the right option to handle a knotty programming problem. The more familiar you become with the commands described in the manual pages the more you will be able to take full advantage of the UNIX system environment.

It is not our purpose here to instruct you in shell programming. What we want to stress here is the important part that shell procedures can play in developing prototypes of full-scale applications. While understanding all the nuances of shell programming can be a fairly complex task, getting a shell procedure up and running is far less time-consuming than writing, compiling and debugging compiled code.

This ability to get a program into production quickly is what makes the shell a valuable tool for program development. Shell programming allows you to "build on the work of others" to the greatest possible degree, since it allows you to piece together major components simply and efficiently. Many times even large applications can be done using shell procedures. Even if the application is initially developed as a prototype system for testing purposes rather than being put into production, many months of work can be saved.

With a prototype for testing, the range of possible user errors can be determined — something that is not always easy to plan out when an application is being designed. The method of dealing with strange user input can be worked out inexpensively, avoiding large re-coding problems.

A common occurrence in the UNIX system environment is to find that an available UNIX system tool can accomplish with a couple of lines of instructions what might take a page and a half of compiled code. Shell procedures can intermix compiled modules and regular UNIX system commands to let you take advantage of work that has gone before.

Three Programming Environments

We distinguish among three programming environments to emphasize that the information needs and the way in which UNIX system tools are used differ from one environment to another. We do not intend to imply a hierarchy of skill or experience. Highly-skilled programmers with years of experience can be found in the "single-user" category, and relative newcomers can be members of an application development or systems programming team.

Single-User Programmer

Programmers in this environment are writing programs only to ease the performance of their primary job. The resulting programs might well be added to the stock of programs available to the community in which the programmer works. This is similar to the atmosphere in which the UNIX system thrived; someone develops a useful tool and shares it with the rest of the organization. Single-user programmers may not have externally imposed requirements, or co-authors, or project management concerns. The programming task itself drives the coding very directly. One advantage of a timesharing system such as UNIX is that people with programming skills can be set free to work on their own without having to go through formal project approval channels and perhaps wait for months for a programming department to solve their problems.

Single-user programmers need to know how to:

- select an appropriate language
- compile and run programs
- use system libraries
- analyze programs

Three Programming Environments

- debug programs
- keep track of program versions

Most of the information to perform these functions at the single-user level can be found in Chapter 2.

Application Programming

Programmers working in this environment are developing systems for the benefit of other, non-programming users. Most large commercial computer applications still involve a team of applications development programmers. They may be employees of the end-user organization or they may work for a software development firm. Some of the people working in this environment may be more in the project management area than working programmers.

Information needs of people in this environment include all the topics in Chapter 2, plus additional information on:

- software control systems
- file and record locking
- communication between processes
- shared memory
- advanced debugging techniques

These topics are discussed in Chapter 3.

Systems Programmers

These are programmers engaged in writing software tools that are part of, or closely related to the operating system itself. The project may involve writing a new device driver, a data base management system or an enhancement to the UNIX system kernel. In addition to knowing their way around the operating system source code and how to make changes and enhancements to it, they need to be thoroughly familiar with all the topics covered in Chapters 2 and 3.

Summary

In this overview chapter we have described the way that the UNIX system developed and the effect that has on the way programmers now work with it. We have described what is and is not to be found in the other chapters of this guide to help programmers. We have also suggested that in many cases programming problems may be easily solved by taking advantage of the UNIX system interactive command interpreter known as the shell. Finally, we identified three programming environments in the hope that it will help orient the reader to the organization of the text in the remaining chapters.

Chapter 2: Programming Basics

	Page
Introduction.....	2- 1
Choosing a Programming Language.....	2- 3
Supported Languages in a System V Environment.....	2- 4
C Language.....	2- 4
FORTRAN.....	2- 5
Pascal.....	2- 6
COBOL.....	2- 6
BASIC.....	2- 6
COMAL 80.....	2- 7
Assembly Language.....	2- 7
Special Purpose Languages.....	2- 7
awk	2- 8
lex	2- 9
yacc	2- 9
M4	2- 9
bc and dc	2- 9
curses	2-10
After Your Code Is Written.....	2-11
Compiling and Link Editing.....	2-12
Compiling C Programs.....	2-12
Loading and Running BASIC Programs.....	2-13
Compiler Diagnostic Messages.....	2-13
Link Editing.....	2-14

Chapter 2: Programming Basics

	Page
The Interface Between a Programming Language and the UNIX System	2-17
Why C Is Used to Illustrate the Interface	2-17
How Arguments Are Passed to a Program.....	2-18
System Calls and Subroutines.....	2-21
Categories of System Calls and Subroutines.....	2-22
Where the Manual Pages Can Be Found.....	2-29
How System Calls and Subroutines Are Used in C Programs	2-29
Header Files and Libraries.....	2-35
Object File Libraries	2-36
Input/Output	2-37
Three Files You Always Have.....	2-38
Named Files	2-39
Low-level I/O and Why You Shouldn't Use It.....	2-40
System Calls for Environment or Status Information	2-42
Processes	2-43
system (3S)	2-45
exec (2)	2-45
fork (2).....	2-46
Pipes	2-48
Error Handling	2-50
Signals and Interrupts	2-50
Analysis/Debugging.....	2-53
Sample Program	2-53
cflow	2-57
ctrace	2-60
cxref	2-64

Chapter 2: Programming Basics

	Page
lint	2-70
size	2-71
strip	2-71
sdb/dbx	2-72
Program Organizing Utilities.....	2-73
The make Command.....	2-73
The Archive.....	2-75
Use of SCCS by Single-User Programmers	2-81

Chapter 2: Programming Basics

This page is intentionally left blank

Introduction

The information in this chapter is for anyone just learning to write programs to run in a UNIX system environment. In Chapter 1 we identified one group of UNIX system users as single-user programmers. People in that category, particularly those who are not deeply interested in programming, may find this chapter (plus related reference manuals) tells them as much as they need to know about coding and running programs on a UNIX system computer.

Programmers whose interest does run deeper, who are part of an application development project, or who are producing programs on one UNIX system computer that are being ported to another, should view this chapter as a starter package.

Introduction

This page is intentionally left blank

Choosing a Programming Language

How do you decide which programming language to use in a given situation? One answer could be, "I always code in HAIRBOL, because that's the language I know best." Actually, in some circumstances that's a legitimate answer. But assuming more than one programming language is available to you, that different programming languages have their strengths and weaknesses, and assuming that once you've learned to use one programming language it becomes relatively easy to learn to use another, you might approach the problem of language selection by asking yourself questions like the following:

- What is the nature of the task this program is to do?
Does the task call for the development of a complex algorithm, or is this a simple procedure that has to be done on a lot of records?
- Does the programming task have many separate parts?
Can the program be subdivided into separately compilable functions, or is it one module?
- How soon does the program have to be available?
Is it needed right now, or do I have enough time to work out the most efficient process possible?
- What is the scope of its use?
Am I the only person who will use this program, or is it going to be distributed to the whole world?
- Is there a possibility the program will be ported to other systems?
- What is the life-expectancy of the program?
Is it going to be used just a few times, or will it still be going strong five years from now?

Supported Languages in a System V Environment

By "supported languages" we mean those offered for use on an SUPERMAX Computer running System V Release 3.1. Since these are separately purchasable items, not all of them will necessarily be installed on your machine. On the other hand, you may have languages available on your machine that came from another source and are not mentioned in this discussion. Be that as it may, in this section and the one to follow we give brief descriptions of the nature of a) six full-scale programming languages, and b) a number of special purpose languages.

C Language

C is intimately associated with the UNIX system since it was originally developed for use in recoding the UNIX system kernel. If you need to use a lot of UNIX system function calls for low-level I/O, memory or device management, or inter-process communication, C language is a logical first choice. Most programs, however, don't require such direct interfaces with the operating system so the decision to choose C might better be based on one or more of the following characteristics:

- a variety of data types: character, integer, long integer, float, and double
- low level constructs (most of the UNIX system kernel is written in C)
- derived data types such as arrays, functions, pointers, structures and unions
- multi-dimensional arrays
- scaled pointers, and the ability to do pointer arithmetic

- bit-wise operators
- a variety of flow-of-control statements: if, if-else, switch, while, do-while, and for
- a high degree of portability

C is a language that lends itself readily to structured programming. It is natural in C to think in terms of functions. The next logical step is to view each function as a separately compilable unit. This approach (coding a program in small pieces) eases the job of making changes and/or improvements. If this begins to sound like the UNIX system philosophy of building new programs from existing tools, it's not just coincidence. As you create functions for one program you will surely find that many can be picked up, or quickly revised, for another program.

A difficulty with C is that it takes a fairly concentrated use of the language over a period of several months to reach your full potential as a C programmer. If you are a casual programmer, you might make life easier for yourself if you choose a less demanding language.

FORTRAN

The oldest of the high-level programming languages, FORTRAN is still highly prized for its variety of mathematical functions. If you are writing a program for statistical analysis or other scientific applications, FORTRAN is a good choice. An original design objective was to produce a language with good operating efficiency. This has been achieved at the expense of some flexibility in the area of type definition and data abstraction. There is, for example, only a single form of the iteration statement. FORTRAN also requires using a somewhat rigid format for input of lines of source code. This shortcoming may be overcome by using one of the UNIX system tools designed to make FORTRAN more flexible.

Pascal

Originally designed as a teaching tool for block structured programming, Pascal has gained quite a wide acceptance because of its straightforward style. Pascal is highly structured and allows system level calls (characteristics it shares with C). Since the intent of the developers, however, was to produce a language to teach people about programming it is perhaps best suited to small projects. Among its inconveniences are its lack of facilities for specifying initial values for variables and limited file processing capability.

COBOL

Probably more programmers are familiar with COBOL than with any other single programming language. It is frequently used in business applications because its strengths lie in the management of input/output and in defining record layouts.

It is somewhat cumbersome to use COBOL for complex algorithms, but it works well in cases where many records have to be passed through a simple process; a payroll withholding tax calculation, for example. It is a rather tedious language to work with because each program requires a lengthy amount of text merely to describe record layouts, processing environment and variables used in the code. The COBOL language is wordy so the compilation process is often quite complex. Once written and put into production, COBOL programs have a way of staying in use for years, and what might be thought of by some as wordiness comes to be considered self-documentation. The investment in programmer time often makes them resistant to change.

44

BASIC

The most commonly heard comment about BASIC is that it is easy to learn. With the spread of personal microcomputers many people have learned BASIC because it is simple to produce runnable programs in very little time. It is difficult, however, to use BASIC for large programming projects. It lacks the provision for structured flow-of-

control, requires that every variable used be defined for the entire program and has no way of transferring values between functions and calling programs. Most versions of BASIC run as interpreted code rather than compiled. That makes for slower running programs. Despite its limitations, however, it is useful for getting simple procedures into operation quickly.

COMAL 80

COMAL 80 is a programming language originally developed for education. COMAL 80 combines the simplicity of the environment of BASIC with the elegant structure of PASCAL. COMAL 80 is easy to learn, and programs written in COMAL 80 are easy to develop. COMAL 80 is an interpreted language making programs slower and therefore not suitable for time consuming tasks.

Assembly Language

The closest approach to machine language, assembly language is specific to the particular computer on which your program is to run. High-level languages are translated into the assembly language for a specific processor as one step of the compilation. The most common need to work in assembly language arises when you want to do some task that is not within the scope of a high-level language. Since assembly language is machine-specific, programs written in it are not portable.

Special Purpose Languages

In addition to the above formal programming languages, the UNIX system environment frequently offers one or more of the special purpose languages listed below.

Language Selection

NOTE

Since UNIX system utilities and commands are packaged in functional groupings, it is possible that not all the facilities mentioned will be available on all systems.

awk

awk (its name is an acronym constructed from the initials of its developers) scans an input file for lines that match pattern(s) described in a specification file. On finding a line that matches a pattern, **awk** performs actions also described in the specification. It is not uncommon that an **awk** program can be written in a couple of lines to do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a quantity. You have sorted the records by the key field, and you now want to add the quantities for records with duplicate keys and output a file in which no keys are duplicated. The pseudo-code for such a program might look like this:

```
Read the first record into a hold area;
Read additional records until EOF;
{
  If the key matches the key of the record in the hold area,
  add the quantity to the quantity field of the held record;
  If the key does not match the key of the held record,
  write the held record,
  move the new record to the hold area;
}
```

At EOF, write out the last record from the hold area.

An **awk** program to accomplish this task would look like this:

```
{ qty[$1] += $2 }
END { for (key in qty) print key, qty[key] }
```

This illustrates only one characteristic of **awk**; its ability to work with associative arrays. With **awk**, the input file does not have to be sorted, which is a requirement of the pseudo-program.

lex

lex is a lexical analyzer that can be added to C programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. **lex** can produce C language subroutines that recognize regular expressions specified by the user, take some action when a regular expression is recognized and pass the output stream on to the next program.

yacc

yacc (Yet Another Compiler Compiler) is a tool for describing an input language to a computer program. **yacc** produces a C language subroutine that parses an input stream according to rules laid down in a specification file. The **yacc** specification file establishes a set of grammar rules together with actions to be taken when tokens in the input match the rules. **lex** may be used with **yacc** to control the input process and pass tokens to the parser that applies the grammar rules.

M4

M4 is a macro processor that can be used as a preprocessor for assembly language, and C programs. It is described in Section (1) of the *System V Reference Manual*.

bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can edit a file of mathematical computations and call **bc** to execute them. The **bc** program uses **dc**. You can use **dc** directly, if you want, but it takes a little getting used to since it works with reverse Polish notation. That means you enter numbers into a stack followed by the operator. **bc** and **dc** are described in Section (1) of the *System V Reference Manual*.

Language Selection

curses

Actually a library of C functions, **curses** is included in this list because the set of functions just about amounts to a sub-language for dealing with terminal screens. If you are writing programs that include interactive user screens, you will want to become familiar with this group of functions.

In addition to all the foregoing, don't overlook the possibility of using shell procedures.

After Your Code Is Written

The last two steps in most compilation systems in the UNIX system environment are the assembler and the link editor. The compilation system produces assembly language code. The assembler translates that code into the machine language of the computer the program is to run on. The link editor resolves all undefined references and makes the object module executable. With most languages on the UNIX system the assembler and link editor produce files in what is known as the Common Object File Format (COFF). A common format makes it easier for utilities that depend on information in the object file to work on different machines running different versions of the UNIX system. In the Common Object File Format an object file contains:

- a file header
- optional secondary header
- a table of section headers
- data corresponding to the section header(s)
- relocation information
- line numbers
- a symbol table
- a string table

An object file is made up of sections. Usually, there are at least two: **.text**, and **.data**. Some object files contain a section called **.bss**. (**.bss** is an assembly language pseudo-op that originally stood for "block started by symbol.") **.bss**, when present, holds uninitialized data. Options of the compilers cause different items of information to be included in the Common Object File Format. Compiling a program with the **-g** option adds line numbers and other symbolic information that is needed for the **sdb** (Symbolic Debugger) command to be fully effective. You can spend many years programming without having to worry too much about the contents and organization of the Common Object File Format, so we are not going into any further depth of detail at this point. See Chapter 10 of this guide.

Compiling and Link Editing

The command used for compiling depends on the language used;

- for C programs, **cc** both compiles and link edits

Compiling C Programs

To use the C compilation system you must have your source code in a file with a filename that ends in the characters **.c**, as in **mycode.c**. The command to invoke the compiler is:

```
cc mycode.c
```

If the compilation is successful the process proceeds through the link edit stage and the result will be an executable file by the name of **a.out**.

Several options to the **cc** command are available to control its operation. The most used options are:

- **c** causes the compilation system to suppress the link edit phase. This produces an object file (**mycode.o**) that can be link edited at a later time with a **cc** command without the **-c** option.
- **g** causes the compilation system to generate special information about variables and language statements used by the symbolic debugger **sdb/dbx**. If you are going through the stage of debugging your program, use this option.
- **O** causes the inclusion of an additional optimization phase. This option is logically incompatible with the **-g** option. You would normally use **-O** after the program has been debugged, to reduce the size of the object file and increase execution speed.

- o *outfile* tells **cc** to tell the link editor to use the specified name for the executable file, rather than the default **a.out**.

Other options can be used with **cc**. Check the *System V Reference Manual*. If you enter the **cc** command using a file name that ends in **.s**, the compilation system treats it as assembly language source code and bypasses all the steps ahead of the assembly step.

As a heterogeneous SUPERMAX may consist of more than one type of processor, a program like **cc**, through the environment parameter **TARGETMC**, is told which cpu to generate code for.

Loading and Running other Programs

The other programming languages have different procedures for compilation and/or interpretation. You are kindly requested to read about this in corresponding manuals.

Compiler Diagnostic Messages

The C compiler generates error messages for statements that don't compile. The messages are quite understandable, but in common with most language compilers they sometimes point several statements beyond where the actual error occurred. If you inadvertently put an extra **;** at the end of an **if** statement, a subsequent **else** will be flagged as a syntax error. In the case where a block of several statements follows the **if**, the line number of the syntax error caused by the **else** will start you looking for the error well past where it is. Unbalanced curly braces, **{ }**, are another common producer of syntax errors.

Link Editing

The **ld** command invokes the link editor directly. The typical user, however, seldom invokes **ld** directly. A more common practice is to use a language compilation control command (such as **cc**) that invokes **ld**.

The link editor combines several object files into one, performs relocation, resolves external symbols, incorporates startup routines, and supports symbol table information used by **sdb/dbx**. You may, of course, start with a single object file rather than several. The resulting executable module is left in a file named **a.out**.

Any file named on the **ld** command line that is not an object file (typically, a name ending in **o**) is assumed to be an archive library or a file of link editor directives. The **ld** command has some 16 options. We are going to describe four of them. These options should be fed to the link editor by specifying them on the **cc** command line if you are doing both jobs with the single command, which is the usual case.

- o outfile** provides a name to be used to replace **a.out** as the name of the output file. Obviously, the name **a.out** is of only temporary usefulness. If you know the name you want use to invoke your program, you can provide it here. Of course, it may be equally convenient to do this:

mv a.out progname

giving your program a less temporary name.

- lx** directs the link editor to search a library **libx.a**, where *x* is up to nine characters. For C programs, **libc.a** is automatically searched if the **cc** command is used. The **-lx** option is used to bring in libraries not normally in the search path such as **libm.a**, the math library. The **-lx** option can occur more than once on a command line, with different values for the *x*. A library is searched when its name is encountered, so the

placement of the option on the command line is important. The safest place to put it is at the end of the command line. The `-lx` option is related to the `-L` option.

`-L dir`

changes the `libx.a` search sequence to search in the specified directory before looking in the default library directories, usually `/lib?` or `/usr/lib?`, where `?` is the TARGETMC. This is useful if you have different versions of a library and you want to point the link editor to the correct one. It works on the assumption that once a library has been found no further searching for that library is necessary. Because `-L` diverts the search for the libraries specified by `-lx` options, it must precede such options on the command line.

`-u symname`

enters `symname` as an undefined symbol in the symbol table. This is useful if you are loading entirely from an archive library, because initially the symbol table is empty and needs an unresolved reference to force the loading of the first routine.

As a heterogeneous SUPERMAX may consist of more than one type of processor, a program like `ld`, through the environment parameter TARGETMC, is told which cpu to generate code for. Different types of executable modules (different TARGETMC) may be put together in one loadmodule by use of the `mkhem(1)` command in order to have one executable module that can run on either processor type.

When the link editor is called through `cc`, a startup routine (typically `/lib?/crt0.o` for C programs, where `?` is the TARGETMC) is linked with your program. This routine calls `exit(2)` after execution of the main program.

Compiling and Link Editing

The link editor accepts a file containing link editor directives. The details of the link editor command language can be found in Chapter 11.

The Interface Between a Programming Language and the UNIX System

When a program is run in a computer it depends on the operating system for a variety of services. Some of the services such as bringing the program into main memory and starting the execution are completely transparent to the program. They are, in effect, arranged for in advance by the link editor when it marks an object module as executable. As a programmer you seldom need to be concerned about such matters.

Other services, however, such as input/output, file management, storage allocation do require work on the part of the programmer. These connections between a program and the UNIX operating system are what is meant by the term UNIX system/language interface. The topics included in this section are:

- How arguments are passed to a program
- System calls and subroutines
- Header files and libraries
- Input/Output
- Processes
- Error Handling, Signals, and Interrupts

Why C Is Used to Illustrate the Interface

Throughout this section C programs are used to illustrate the interface between the UNIX system and programming languages because C programs make more use of the interface mechanisms than other high-level languages. What is really being covered in this section then is the UNIX system/C Language interface. The way that other languages deal with these topics is described in the user's guides for those languages.

How Arguments Are Passed to a Program

Information or control data can be passed to a C program as arguments on the command line. When the program is run as a command, arguments on the command line are made available to the function **main** in two parameters, an argument count and an array of pointers to character strings. (Every C program is required to have an entry module by the name of **main**.) Since the argument count is always given, the program does not have to know in advance how many arguments to expect. The character strings pointed at by elements of the array of pointers contain the argument information.

The arguments are presented to the program traditionally as **argc** and **argv**, although any names you choose will work. **argc** is an integer that gives the count of the number of arguments. Since the command itself is considered to be the first argument, **argv[0]**, the count is always at least one. **argv** is an array of pointers to character strings (arrays of characters terminated by the null character `\0`).

If you plan to pass runtime parameters to your program, you need to include code to deal with the information. Two possible uses of runtime parameters are:

- as control data. Use the information to set internal flags that control the operation of the program.
- to provide a variable filename to the program.

Figures 2-1 and 2-2 show program fragments that illustrate these uses.


```
#include <stdio.h>
main(argc, argv)
    int argc;
    char *argv[];
{
    void exit();
    int oflag = FALSE;
    int pflag = FALSE;    /* Function Flags */
    int rflag = FALSE;
    int ch;

    while ((ch = getopt(argc,argv, "opr")) != EOF)
    {
        /* For options present, set flag to TRUE */
        /* If no options present, print error message */
        switch (ch)
        {
            case 'o':
                oflag = 1;
                break;
            case 'p':
                pflag = 1;
                break;
            case 'r':
                rflag = 1;
                break;
            default:
                (void)fprintf(stderr,
                    "Usage: %s [-opr]\n", argv[0]);
                exit(2);
        }
    }
}
```

Figure 2-1: Using Command Line Arguments to Set Flags

```
#include <stdio.h>
main(argc, argv)
  int argc;
  char *argv[];
{
    FILE *fopen(), *fin;
    void perror(), exit();
    if (argc > 1)
    {
        if ((fin = fopen(argv[1], "r")) == NULL)
        {
            /* First string (%s) is program name (argv[0]) */
            /* Second string (%s) is name of file that */
            /* could not be opened (argv[1]) */
            (void)fprintf(stderr,
                "%s: cannot open %s: ",
                argv[0], argv[1]);
            perror("");
            exit(2);
        }
    }
    .
    .
}
```

Figure 2-2: Using `arg[n]` Pointers to Pass a Filename

The shell, which makes arguments available to your program, considers an argument to be any non-blank characters separated by blanks or tabs. Characters enclosed in double quotes ("abc def") are passed to the program as one argument even if blanks or tabs are among the characters. It goes without saying that you are responsible for error checking and otherwise making sure the argument received is what your program expects it to be.

A third argument is also present, in addition to **argc** and **argv**. The third argument, known as **envp**, is an array of pointers to environment variables. You can find more information on **envp** in the *System V Reference Manual* under **exec(2)** and **environ(5)**.

System Calls and Subroutines

System calls are requests from a program for an action to be performed by the UNIX system kernel. Subroutines are precoded modules used to supplement the functionality of a programming language.

Both system calls and subroutines look like functions such as those you might code for the individual parts of your program. There are, however, differences between them:

- At link edit time, the code for subroutines is copied into the object file for your program; the code invoked by a system call remains in the kernel.
- At execution time, subroutine code is executed as if it was code you had written yourself; a system function call is executed by switching from your process area to the kernel.

This means that while subroutines make your executable object file larger, runtime overhead for context switching may be less and execution may be faster.

Categories of System Calls and Subroutines

System calls divide fairly neatly into the following categories:

- file access
- file and directory manipulation
- process control
- environment control and status information

You can generally tell the category of a subroutine by the section of the *System V Reference Manual* in which you find its manual page. However, the first part of Section 3 (3C and 3S) covers such a variety of subroutines it might be helpful to classify them further.

- The subroutines of sub-class 3S constitute the UNIX system/C Language standard I/O, an efficient I/O buffering scheme for C.
- The subroutines of sub-class 3C do a variety of tasks. They have in common the fact that their object code is stored in **libc.a**. They can be divided into the following categories:
 - string manipulation
 - character conversion
 - character classification
 - environment management
 - memory management

Figure 2-3 lists the functions that compose the standard I/O subroutines. Frequently, one manual page describes several related functions. In Figure 2-3 the left hand column contains the name that appears at the top of the manual page; the other names in the same row are related functions described on the same manual page.

Function Name(s)				Purpose
fclose	fflush			close or flush a stream
ferror	feof	clearerr	fileno	stream status inquiries
fopen	freopen	fdopen		open a stream
fread	fwrite			binary input/output
fseek	rewind	ftell		reposition a file pointer in a stream
getc	getchar	fgetc	getw	get a character or word from a stream
gets	fgets			get a string from a stream
popen	pclose			begin or end a pipe to or from a process
printf	fprintf	sprintf		print formatted output

For all functions: #include <stdio.h>

The function name shown in **bold** gives the location in the *System V Reference Manual*, Section 3.

Figure 2-3: C Language Standard I/O Subroutines (sheet 1 of 2)

The UNIX System/Language Interface

Function Name(s)				Purpose
putc	putchar	fputc	putw	put a character or word on a stream
puts	fputs			put a string on a stream
scanf	fscanf	sscanf		convert formatted input
setbuf	setvbuf			assign buffering to a stream
system				issue a command through the shell
tmpfile				create a temporary file
tmpnam		tmpnam		create a name for a temporary file
ungetc				push character back into input stream
vprintf	vfprintf	vsprintf		print formatted output of a varargs argument list

For all functions: `#include <stdio.h>`

The function name shown in **bold** gives the location in the *System V Reference Manual*, Section 3.

Figure 2-3: C Language Standard I/O Subroutines (sheet 2 of 2)

Figure 2-4 lists string handling functions that are grouped under the heading **string(3C)** in the *System V Reference Manual*.

String Operations

strcat(s1, s2)	append a copy of s2 to the end of s1.
strncat(s1, s2, n)	append n characters from s2 to the end of s1.
strcmp(s1, s2)	compare two strings. Returns an integer less than, greater than or equal to 0 to show that s1 is lexicographically less than, greater than or equal to s2.
strncmp(s1, s2, n)	compare n characters from the two strings. Results are otherwise identical to strcmp.
strcpy(s1, s2)	copy s2 to s1, stopping after the null character (\0) has been copied.
strncpy(s1, s2, n)	copy n characters from s2 to s1. s2 will be truncated if it is longer than n, or padded with null characters if it is shorter than n.
strdup(s)	returns a pointer to a new string that is a duplicate of the string pointed to by s.
strchr(s, c)	returns a pointer to the first occurrence of character c in string s, or a NULL pointer if c is not in s.
strrchr(s, c)	returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c is not in s.

For all functions: #include <string.h>

string.h provides extern definitions of the string functions.

Figure 2-4: String Operations (sheet 1 of 2)

String Operations

strlen(s)	returns the number of characters in s up to the first null character.
strpbrk(s1, s2)	returns a pointer to the first occurrence in s1 of any character from s2, or a NULL pointer if no character from s2 occurs in s1.
strspn(s1, s2)	returns the length of the initial segment of s1, which consists entirely of characters from s2.
strcspn(s1, s2)	returns the length of the initial segment of s1, which consists entirely of characters not from s2.
strtok(s1, s2)	look for occurrences of s2 within s1.

For all functions: #include <string.h>

string.h provides extern definitions of the string functions.

Figure 2-4: String Operations (sheet 2 of 2)

Figure 2-5 lists macros that classify ASCII character-coded integer values. These macros are described under the heading **ctype(3C)** in Section 3 of the *System V Reference Manual*.

Classify Characters

isalpha(c)	is <i>c</i> a letter
isupper(c)	is <i>c</i> an upper-case letter
islower(c)	is <i>c</i> a lower-case letter
isdigit(c)	is <i>c</i> a digit [0-9]
isxdigit(c)	is <i>c</i> a hexadecimal digit [0-9], [A-F] or [a-f]
isalnum(c)	is <i>c</i> an alphanumeric (letter or digit)
isspace(c)	is <i>c</i> a space, tab, carriage return, new-line, vertical tab or form-feed
ispunct(c)	is <i>c</i> a punctuation character (neither control nor alphanumeric)
isprint(c)	is <i>c</i> a printing character, code 040 (space) through 0176 (tilde) and 0240 (NBSP) through 0377 (small letter y with direses).
isgraph(c)	same as isprint except false for 040 (space)
iscntrl(c)	is <i>c</i> a control character (less than 040) or a delete character (0177)
isascii(c)	is <i>c</i> an ASCII character (code less than 0200)

For all macros: #include <ctype.h>
 Nonzero return == true; zero return == false

Figure 2-5: Classifying ASCII Character-Coded Integer Values

The UNIX System/Language Interface

Figure 2-6 lists functions and macros that are used to convert characters, integers, or strings from one representation to another.

Function Name(s)			Purpose
a64l	l64a		convert between long integer and base-64 ASCII string
ecvt	fcvt	gcvt	convert floating-point number to string
l3tol	l3tol		convert between 3-byte integer and long integer
strtod	atof		convert string to double-precision number
strtol	atol	atoi	convert string to integer
conv(3C):			Translate Characters
toupper			lower-case to upper-case
_toupper			macro version of toupper
tolower			upper-case to lower-case
_tolower			macro version of tolower
toascii			turn off all bits that are not part of a standard ASCII character; intended for compatibility with other systems

For all **conv(3C)** macros: `#include <ctype.h>`

Figure 2-6: Conversion Functions and Macros

Where the Manual Pages Can Be Found

System calls are listed alphabetically in Section 2 of the *System V Reference Manual*. Subroutines are listed in Section 3. We have described above what is in the first subsection of Section 3. The remaining subsections of Section 3 are:

- 3M – functions that make up the Math Library, **libm**
- 3X – various specialized functions
- 3N – Networking Support Utilities

How System Calls and Subroutines Are Used in C Programs

Information about the proper way to use system calls and subroutines is given on the manual page, but you have to know what you are looking for before it begins to make sense. To illustrate, a typical manual page (for **gets(3S)**) is shown in Figure 2-7 on the following page.

The UNIX System/Language Interface

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>
char * gets (s)
    char * s;

char * fgets (s, n, stream)
    char * s;
    int n;
    FILE * stream;
```

DESCRIPTION

gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

fgets reads characters from the *stream* into the array pointed to by *s*, until $n - 1$ characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

error(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

Figure 2-7: Manual Page for **gets(3S)**

As you can see from the illustration, two related functions are described on this page: **gets** and **fgets**. Each function gets a string from a stream in a slightly different way. The DESCRIPTION section tells how each operates.

It is the SYNOPSIS section, however, that contains the critical information about how the function (or macro) is used in your program. Notice in Figure 2-7 that the first line in the SYNOPSIS is

```
#include <stdio.h>
```

This means that to use **gets** or **fgets** you must bring the standard I/O header file into your program (generally right at the top of the file). There is something in **stdio.h** that is needed when you use the described functions. Figure 2-9 shows a version of **stdio.h**. Check it to see if you can understand what **gets** or **fgets** uses.

The next thing shown in the SYNOPSIS section of a manual page that documents system calls or subroutines is the formal declaration of the function. The formal declaration tells you:

- **the type of object returned by the function**

In our example, both **gets** and **fgets** return a character pointer.

- **the object or objects the function expects to receive when called**

These are the things enclosed in the parentheses of the function. **gets** expects a character pointer. (The DESCRIPTION section sheds light on what the tokens of the formal declaration stand for.)

- **how the function is going to treat those objects**

The declaration

```
char *s;
```

in **gets** means that the token **s** enclosed in the parentheses will be considered to be a pointer to a character string. Bear in mind that in the C language, when passed as an argument, the

The UNIX System/Language Interface

name of an array is converted to a pointer to the beginning of the array.

We have chosen a simple example here in **gets**. If you want to test yourself on something a little more complex, try working out the meaning of the elements of the **fgets** declaration.

While we're on the subject of **fgets**, there is another piece of C esoterica that we'll explain. Notice that the third parameter in the **fgets** declaration is referred to as **stream**. A **stream**, in this context, is a file with its associated buffering. It is declared to be a pointer to a defined type **FILE**. Where is **FILE** defined? Right! In **stdio.h**.

To finish off this discussion of the way you use functions described in the *System V Reference Manual* in your own code, in Figure 2-??? we show a program fragment in which **gets** is used.

```
#include <stdio.h>

main()
{
    char sarray[80];

    for(;;)
    {
        if (gets(sarray) != NULL)
            .
            .      /* Do something with the string */
            .
    }
}
```

Figure 2-8: How **gets** Is Used in a Program

You might ask, "Where is **gets** reading from?" The answer is, "From the standard input." That generally means from something being keyed in from the terminal where the command was entered to get the program running, or output from another command that was piped to **gets**. How do we know that? The **DESCRIPTION** section of

the **gets** manual page says, "gets reads characters from the standard input...." Where is the standard input defined? In **stdio.h**.

```
#ifndef _NFILE
#define _NFILE 32

#define BUFSIZ 2048
#define _SBFSIZ 8

typedef struct {
    int          _cnt;
    unsigned char *_ptr;
    unsigned char *_base;
    char         _flag;
    char         _file;
} FILE;

#define _IOFBF    0000 /* _IOLBF means that a file's output */
#define _IOREAD  0001 /* will be buffered line by line. */
#define _IOWRT   0002 /* In addition to being flags, _IONBF,*/
#define _IONBF   0004 /* _IOLBF and IOFBF are possible */
#define _IOMYBUF 0010 /* values for "type" in setvbuf. */
#define _IOEOF   0020
#define _IOERR   0040
#define _IOLBF   0100
#define _IORW    0200

#ifndef NULL
#define NULL 0
#endif
#ifndef EOF
#define EOF (-1)
#endif
```

Figure 2-9: A Version of **stdio.h** (sheet 1 of 2)

```

#define stdin          (&_iob[0])
#define stdout         (&_iob[1])
#define stderr        (&_iob[2])

#define _bufend(p)    _bufendtab[(p)->_file]
#define _bufsiz(p)   (_bufend(p) - (p)->_base)

#ifdef lint
#define getc(p)       (--(p)->_cnt < 0 ? _filbuf(p) : (int) *(p)->_ptr++)
#define putc(x, p)   (--(p)->_cnt < 0 ?
                    _flsbuf((unsigned char) (x), (p)) :
                    (int) *(p)->_ptr++ = (unsigned char) (x))
#define getchar()    getc(stdin)
#define putchar(x)  putc((x), stdout)
#define clearerr(p) ((void) ((p)->_flag &= ~(_IOERR | _IOEOF)))
#define feof(p)     ((p)->_flag & _IOEOF)
#define ferror(p)  ((p)->_flag & _IOERR)
#define fileno(p)  (p)->_file
#endif

extern FILE _iob[_NFILE];
extern FILE *fopen(), *fdopen(), *freopen(), *popen(), *tmpfile();
extern long ftell();
extern void rewind(), setbuf();
extern char *ctermid(), *cuserid(), *fgets(), *gets(), *tempnam(), *tmpnam();
extern unsigned char *_bufendtab[];

#define L_ctermid 9
#define L_cuserid 9
#define P_tmpdir  "/usr/tmp/"
#define L_tmpnam  (sizeof(P_tmpdir) + 15)
#endif

```

72

Figure 2-9: A Version of **stdio.h** (sheet 2 of 2)

Header Files and Libraries

In the earlier parts of this chapter there have been frequent references to **stdio.h**, and a version of the file itself is shown in Figure 2-9. **stdio.h** is the most commonly used header file in the UNIX system/C environment, but there are many others.

Header files carry definitions and declarations that are used by more than one function. Header filenames traditionally have the suffix **.h**, and are brought into a program at compile time by the C-preprocessor. The preprocessor does this because it interprets the **#include** statement in your program as a directive; as indeed it is. All keywords preceded by a pound sign (**#**) at the beginning of the line, are treated as preprocessor directives. The two most commonly used directives are **#include** and **#define**. We have already seen that the **#include** directive is used to call in (and process) the contents of the named file. The **#define** directive is used to replace a name with a token-string. For example,

```
#define _NFILE 32
```

sets to 32 the number of files a program can have open at one time. See **cpp(1)** for the complete list.

In the pages of the *System V Reference Manual* there are about 45 different **.h** files named. The format of the **#include** statement for all these shows the file name enclosed in angle brackets (**<>**), as in

```
#include <stdio.h>
```

The angle brackets tell the C preprocessor to look in the standard places for the file. In most systems the standard place is in the **/usr/include** directory. If you have some definitions or external declarations that you want to make available in several files, you can create a **.h** file with any editor, store it in a convenient directory and make it the subject of a **#include** statement such as the following:

```
#include "../defs/rec.h"
```

It is necessary, in this case, to provide the relative pathname of the file and enclose it in quotation marks (""). Fully-qualified pathnames (those that begin with /) can create portability and organizational problems. An alternative to long or fully-qualified pathnames is to use the `-Idir` preprocessor option when you compile the program. This option directs the preprocessor to search for `#include` files whose names are enclosed in "", first in the directory of the file being compiled, then in the directories named in the `-I` option(s), and finally in directories on the standard list. In addition, all `#include` files whose names are enclosed in angle brackets (< >) are first searched for in the list of directories named in the `-I` option and finally in the directories on the standard list.

Object File Libraries

It is common practice in UNIX system computers to keep modules of compiled code (object files) in archives; by convention, designated by a `.a` suffix. System calls from Section 2, and the subroutines in Section 3, subsections 3C and 3S, of the *System V Reference Manual* that are functions (as distinct from macros) are kept in an archive file by the name of `libc.a`. In most systems, `libc.a` is found in the directory `/lib?` or `/usr/lib?`, where ? is the TARGETMC. If both libraries occur, the latter is apt to be used to hold archives that are related to specific applications.

During the link edit phase of the compilation and link edit process, copies of some of the object modules in an archive file are loaded with your executable code. By default the `cc` command that invokes the C compilation system causes the link editor to search `libc.a`. If you need to point the link editor to other libraries that are not searched by default, you do it by naming them explicitly on the command line with the `-l` option. The format of the `-l` option is `-lx` where `x` is the library name, and can be up to nine characters. For example, if your program includes functions from the `curses` screen control package, the option

`-lcurses`

will cause the link editor to search for **/lib?/libcurses.a** or **/usr/lib?/libcurses.a**, (? is the TARGETMC), and use the first one it finds to resolve references in your program.

In cases where you want to direct the order in which archive libraries are searched, you may use the **-L *dir*** option. Assuming the **-L** option appears on the command line ahead of the **-l** option, it directs the link editor to search the named directory for **libx.a** before looking in **/lib?** and **/usr/lib?**, where ? is TARGETMC. This is particularly useful if you are testing out a new version of a function that already exists in an archive in a standard directory. Its success is due to the fact that once having resolved a reference the link editor stops looking. That's why the **-L** option, if used, should appear on the command line ahead of any **-l** specification.

Input/Output

We talked some about I/O earlier in this chapter in connection with system calls and subroutines. A whole set of subroutines constitutes the C language standard I/O package, and there are several system calls that deal with the same area. In this section we want to get into the subject in a little more detail and describe for you how to deal with input and output concerns in your C programs. First off, let's briefly define what the subject of I/O encompasses. It has to do with

- creating and sometimes removing files
- opening and closing files used by your program
- transferring information from a file to your program (reading)
- transferring information from your program to a file (writing)

In this section we will describe some of the subroutines you might choose for transferring information, but the heaviest emphasis will be on dealing with files.

Three Files You Always Have

Programs are permitted to have several files open simultaneously. The number may vary from system to system; the most common maximum is 32. `_NFILE` in `stdio.h` specifies the number of standard I/O FILEs a program is permitted to have open.

Any program automatically starts off with three files. If you will look again at Figure 2-9, about midway through you will see that `stdio.h` contains three `#define` directives that equate `stdin`, `stdout`, and `stderr` to the address of `_iob[0]`, `_iob[1]`, and `_iob[2]`, respectively. The array `_iob` holds information dealing with the way standard I/O handles streams. It is a representation of the open file table in the control block for your program. The position in the array is a digit that is also known as the file descriptor. The default in UNIX systems is to associate all three of these files with your terminal.

The real significance is that functions and macros that deal with `stdin` or `stdout` can be used in your program with no further need to open or close files. For example, `gets`, cited above, reads a string from `stdin`; `puts` writes a null-terminated string to `stdout`. There are others that do the same (in slightly different ways: character at a time, formatted, etc.). You can specify that output be directed to `stderr` by using a function such as `fprintf`. `fprintf` works the same as `printf` except that it delivers its formatted output to a named stream, such as `stderr`. You can use the shell's redirection feature on the command line to read from or write into a named file. If you want to separate error messages from ordinary output being sent to `stdout` and thence possibly piped by the shell to a succeeding program, you can do it by using one function to handle the ordinary output and a variation of the same function that names the stream, to handle error messages.

Named Files

Any files other than **stdin**, **stdout**, and **stderr** that are to be used by your program must be explicitly connected by you before the file can be read from or written to. This can be done using the standard library routine **fopen**. **fopen** takes a pathname (which is the name by which the file is known to the UNIX file system), asks the system to keep track of the connection, and returns a pointer that you then use in functions that do the reads and writes.

A structure is defined in **stdio.h** with a type of **FILE**. In your program you need to have a declaration such as

```
FILE *fin;
```

The declaration says that **fin** is a pointer to a **FILE**. You can then assign the name of a particular file to the pointer with a statement in your program like this:

```
fin = fopen("filename", "r");
```

where **filename** is the pathname to open. The "r" means that the file is to be opened for reading. This argument is known as the **mode**. As you might suspect, there are modes for reading, writing, and both reading and writing. Actually, the file open function is often included in an if statement such as:

```
if ((fin = fopen("filename", "r")) == NULL)
    (void)fprintf(stderr, "%s: Unable to open file %s\n", argv[0], "filename");
```

that takes advantage of the fact that **fopen** returns a **NULL** pointer if it can't open the file.

Once the file has been successfully opened, the pointer **fin** is used in functions (or macros) to refer to the file. For example:

```
int c;
c = getc(fin);
```

brings in a character at a time from the file into an integer variable called **c**. The variable **c** is declared as an integer even though we are reading characters because the function **getc()** returns an integer. Getting a character is often incorporated into some flow-of-control mechanism such as:

```
while ((c = getc(fin)) != EOF)
```

```
    .
    .
    .
```

that reads through the file until EOF is returned. EOF, NULL, and the macro **getc** are all defined in **stdio.h**. **getc** and others that make up the standard I/O package keep advancing a pointer through the buffer associated with the file; the UNIX system and the standard I/O subroutines are responsible for seeing that the buffer is refilled (or written to the output file if you are producing output) when the pointer reaches the end of the buffer. All these mechanics are mercifully invisible to the program and the programmer.

The function **fclose** is used to break the connection between the pointer in your program and the pathname. The pointer may then be associated with another file by another call to **fopen**. This re-use of a file descriptor for a different stream may be necessary if your program has many files to open. For output files it is good to issue an **fclose** call because the call makes sure that all output has been sent from the output buffer before disconnecting the file. The system call **exit** closes all open files for you. It also gets you completely out of your process, however, so it is safe to use only when you are sure you are completely finished.

Low-level I/O and Why You Shouldn't Use It

The term low-level I/O is used to refer to the process of using system calls from Section 2 of the *System V Reference Manual* rather than the functions and subroutines of the standard I/O package. We are going to postpone until Chapter 3 any discussion of when this might be advantageous. If you find as you go through the information in

this chapter that it is a good fit with the objectives you have as a programmer, it is a safe assumption that you can work with C language programs in the UNIX system for a good many years without ever having a real need to use system calls to handle your I/O and file accessing problems. The reason low-level I/O is perilous is because it is more system-dependent. Your programs are less portable and probably no more efficient.

System Calls for Environment or Status Information

Under some circumstances you might want to be able to monitor or control the environment in your computer. There are system calls that can be used for this purpose. Some of them are shown in Figure 2-10.

Function Name(s)			Purpose
chdir			change working directory
chmod			change access permission of a file
chown			change owner and group of a file
getpid	getpgrp	getppid	get process IDs
getuid	geteuid	getgid	get user IDs
ioctl			control device
link	unlink		add or remove a directory entry
mount	umount		mount or unmount a file system
nice			change priority of a process
stat	fstat		get file status
time			get time
ulimit			get and set user limits
uname			get name of current UNIX system

Figure 2-10: Environment and Status System Calls

As shown in Figure 2-10, many of the functions have equivalent UNIX system shell commands. Shell commands can easily be incorporated into shell scripts to accomplish the monitoring and control tasks you may need to do. The functions are available, however, and may be used in C programs as part of the UNIX system/C Language

interface. They are documented in Section 2 of the *System V Reference Manual*.

Processes

Whenever you execute a command in the UNIX system you are initiating a process that is numbered and tracked by the operating system. A flexible feature of the UNIX system is that processes can be generated by other processes. This happens more than you might ever be aware of.

For example, when you log in to your system you are running a process, very probably the shell. If you then use an editor such as **vi**, take the option of invoking the shell from **vi**, and execute the **ps** command, you will see a display something like that in Figure 2-11 (which shows the results of a **ps -f** command):

```

UID  PID  PPID  C  STIME  TTY  TIME  COMMAND
abc  24210  1     0  06:13:14  tty29  0:05  -sh
abc  24631  24210  0  06:59:07  tty29  0:13  vi c2.uli
abc  28441  28358  80  09:17:22  tty29  0:01  ps -f
abc  28358  24631  2  09:15:14  tty29  0:01  sh -i

```

Figure 2-11: Process Status

As you can see, user abc (who went through the steps described above) now has four processes active. It is an interesting exercise to trace the chain that is shown in the Process ID (PID) and Parent Process ID (PPID) columns. The shell that was started when user abc logged on is Process 24210; its parent is the initialization process (Process ID 1). Process 24210 is the parent of Process 24631, and so on.

The four processes in the example above are all UNIX system shell level commands, but you can spawn new processes from your own program.

The UNIX System/Language Interface

(Actually, when you issue the command from your terminal to execute a program you are asking the shell to start another process, the process being your executable object module with all the functions and subroutines that were made a part of it by the link editor.)

You might think, "Well, it's one thing to switch from one program to another when I'm at my terminal working interactively with the computer; but why would a program want to run other programs, and if one does, why wouldn't I just put everything together into one big executable module?"

Overlooking the case where your program is itself an interactive application with diverse choices for the user, your program may need to run one or more other programs based on conditions it encounters in its own processing. (If it's the end of the month, go do a trial balance, for example.) The usual reasons why it might not be practical to create one monster executable are:

- The load module may get too big to fit in the maximum process size for your system.
- You may not have control over the object code of all the other modules you want to include.

Suffice it to say, there are legitimate reasons why this creation of new processes might need to be done. There are three ways to do it:

- **system(3S)** – request the shell to execute a command
- **exec(2)** – stop this process and start another
- **fork(2)** – start an additional copy of this process

system(3S)

The formal declaration of the **system** function looks like this:

```
#include <stdio.h>

int system(string)
char *string;
```

The function asks the shell to treat the string as a command line. The string can therefore be the name and arguments of any executable program or UNIX system shell command. If the exact arguments vary from one execution to the next, you may want to use **sprintf** to format the string before issuing the **system** command. When the command has finished running, **system** returns the shell exit status to your program. Execution of your program waits for the completion of the command initiated by **system** and then picks up again at the next executable statement.

exec(2)

exec is the name of a family of functions that includes **execv**, **execle**, **execve**, **execlp**, and **execvp**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function. An example of one version (**execl**) might be:

```
execl("/bin/prog2", "prog", progarg1, progarg2, (char *)0);
```

For **execl** the argument list is

/bin/prog2	path name of the new process file
prog	the name the new process gets in its argv[0]
progarg1, progarg2	arguments to <i>prog2</i> as char *'s
(char *)0	a null char pointer to mark the end of the arguments

Check the manual page in the *System V Reference Manual* for the rest of the details. The key point of the **exec** family is that there is no return from a successful execution: the calling process is finished, the new process overlays the old. The new process also takes over the Process ID and other attributes of the old process. If the call to **exec** is unsuccessful, control is returned to your program with a return value of -1 . You can check **errno** (see below) to learn why it failed.

fork(2)

The **fork** system call creates a new process that is an exact copy of the calling process. The new process is known as the child process; the caller is known as the parent process. The one major difference between the two processes is that the child gets its own unique process ID. When the **fork** process has completed successfully, it returns a 0 to the child process and the child's process ID to the parent. If the idea of having two identical processes seems a little funny, consider this:

- Because the return value is different between the child process and the parent, the program can contain the logic to determine different paths.
- The child process could say, "Okay, I'm the child. I'm supposed to issue an **exec** for an entirely different program."
- The parent process could say, "My child is going to be **execing** a new process. I'll issue a **wait** until I get word that that process is finished."

To take this out of the storybook world where programs talk like people and into the world of C programming (where people talk like programs), your code might include statements like this:

```
#include <errno.h>

int ch_stat, ch_pid, status;
char *progarg1;
char *progarg2;
void exit();
extern int errno;

    if ((ch_pid = fork()) < 0)
    {
        /* Could not fork...
           check errno
           */
    }
    else if (ch_pid == 0)      /* child */
    {
        (void)execl("/bin/prog2", "prog", progarg1, progarg2, (char *)0);
        exit(2); /* execl() failed */
    }
    else                      /* parent */
    {
        while ((status = wait(&ch_stat)) != ch_pid)
        {
            if (status < 0 && errno == ECHILD)
                break;
            errno = 0;
        }
    }
}
```

Figure 2-12: Example of **fork**

Because the child process ID is taken over by the new **exec**'d process, the parent knows the ID. What this boils down to is a way of leaving one program to run another, returning to the point in the first program where processing left off. This is exactly what the **system(3S)** function does. As a matter of fact, **system** accomplishes it through this same procedure of **forking** and **execing**, with a **wait** in the parent.

The UNIX System/Language Interface

Keep in mind that the fragment of code above includes a minimum amount of checking for error conditions. There is also potential confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the **fork** or **exec** has the three standard files that are automatically opened: **stdin**, **stdout**, and **stderr**.

If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork. Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process the pointer has moved on.

Pipes

The idea of using pipes, a connection between the output of one program and the input of another, when working with commands executed by the shell is well established in the UNIX system environment.

For example, to learn the number of archive files in your system you might enter a command like:

```
echo /lib*/*.a /usr/lib*/*.a | wc -w
```

that first echoes all the files in **/lib?** and **/usr/lib?**, (? is the TARGETMC), that end in **.a**, then pipes the results to the **wc** command, which counts their number.

A feature of the UNIX system/C Language interface is the ability to establish pipe connections between your process and a command to be executed by the shell, or between two cooperating processes. The first uses the **popen(3S)** subroutine that is part of the standard I/O package; the second requires the system call **pipe(2)**.

popen is similar in concept to the **system** subroutine in that it causes the shell to execute a command. The difference is that once having invoked **popen** from your program, you have established an open line to a concurrently running process through a stream. You

Error Handling

Within your C programs you must determine the appropriate level of checking for valid data and for acceptable return codes from functions and subroutines. If you use any of the system calls described in Section 2 of the *System V Reference Manual*, you have a way in which you can find out the probable cause of a bad return value.

UNIX system calls that are not able to complete successfully almost always return a value of `-1` to your program. (If you look through the system calls in Section 2, you will see that there are a few calls for which no return value is defined, but they are the exceptions.) In addition to the `-1` that is returned to the program, the unsuccessful system call places an integer in an externally declared variable, **errno**. You can determine the value in **errno** if your program contains the statement

```
#include <errno.h>
```

The value in **errno** is not cleared on successful calls, so your program should check it only if the system call returned a `-1`. The errors are described in **intro(2)** of the *System V Reference Manual*.

The subroutine **perror(3C)** can be used to print an error message (on **stderr**) based on the value of **errno**.

Signals and Interrupts

Signals and interrupts are two words for the same thing. Both words refer to messages passed by the UNIX system to running processes. Generally, the effect is to cause the process to stop running. Some signals are generated if the process attempts to do something illegal; others can be initiated by a user against his or her own processes, or by the super-user against any process.

There is a system call, **kill**, that you can include in your program to send signals to other processes running under your user-id. The format for the **kill** call is:

```
kill(pid, sig)
```

where **pid** is the process number against which the call is directed, and **sig** is an integer from 1 to 19 that shows the intent of the message. The name "kill" is something of an overstatement; not all the messages have a "drop dead" meaning. Some of the available signals are shown in Figure 2-14 as they are defined in `<sys/signal.h>`.

```
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL 4 /* illegal instruction (not reset when caught) */
#define SIGTRAP 5 /* trace trap (not reset when caught) */
#define SIGIOT 6 /* IOT instruction */
#define SIGABRT 6 /* used by abort, replace SIGIOT in the future */
#define SIGEMT 7 /* EMT instruction */
#define SIGFPE 8 /* floating point exception */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS 12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD 18 /* death of a child */
#define SIGPWR 19 /* power-fail restart */

#define SIGPOLL 22 /* pollable event occurred */

#define NSIG 32 /* maximum number of exceptions */
#define MAXSIG 32 /* maximum number of exceptions */
```

Figure 2-14: Signal Numbers Defined in `/usr/include/sys/signal.h`

The UNIX System/Language Interface

System generated signals are sent to all processes started from the terminal that carries the same process group-id as the process that is really the target. Unless some other provision within the program is made to field the signal, the processes are terminated when an **interrupt**, **quit**, **hangup**, or **terminate** signal is received.

The **signal(2)** system call is designed to let you code methods of dealing with incoming signals. You have a three-way choice. You can a) accept whatever the default action is for the signal, b) have your program ignore the signal, or c) write a function of your own to deal with it.

Analysis/Debugging

The UNIX system provides several commands designed to help you discover the causes of problems in programs and to learn about potential problems.

Sample Program

To illustrate how these commands are used and the type of output they produce, we have constructed a sample program that opens and reads an input file and performs one to three subroutines according to options specified on the command line. This program does not do anything you couldn't do quite easily on your pocket calculator, but it does serve to illustrate some points. The source code is shown in Figure 2-15. The header file, **recdef.h**, is shown at the end of the source code.

The output produced by the various analysis and debugging tools illustrated in this section may vary slightly from one installation to another. The *System V Reference Manual* is a good source of additional information about the contents of the reports.

Figure 2-15: Source Code for Sample Program

```
/* Main module -- restate.c */

#include <stdio.h>
#include "recdef.h"

#define TRUE          1
#define FALSE        0

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fopen(), *fin;
    void exit();
    int getopt();
    int oflag = FALSE;
    int pflag = FALSE;
    int rflag = FALSE;
    int ch;
    struct rec first;
    extern int opterr;
    extern float oppty(), pft(), rfe();

    if (argc < 2)
    {
        (void) fprintf(stderr, "%s: Must specify option\n", argv[0]);
        (void) fprintf(stderr, "Usage: %s -rpo\n", argv[0]);
        exit(2);
    }

    opterr = FALSE;
    while ((ch = getopt(argc, argv, "opr")) != EOF)
    {
        switch(ch)
        {
            case 'o':
                oflag = TRUE;
                break;
            case 'p':
                pflag = TRUE;
                break;
        }
    }
}
```

(continued on next page)

```

    case 'r':
        rflag = TRUE;
        break;
    default:
        (void) fprintf(stderr, "Usage: %s -rpo\n", argv[0]);
        exit(2);
}
}
if ((fin = fopen("info", "r")) == NULL)
{
    (void) fprintf(stderr, "%s: cannot open input file %s\n",
        argv[0], "info");
    exit(2);
}
if (fscanf(fin, "%s%f%f%f%f", first.pname, &first.ppx,
    &first.dp, &first.i, &first.c, &first.t, &first.spx) != 7)
{
    (void) fprintf(stderr, "%s: cannot read first record from %s\n",
        argv[0], "info");
    exit(2);
}

printf("Property: %s\n", first.pname);

if(oflag)
    printf("Opportunity Cost: $%#5.2f\n", oppty(&first));

if(pflag)
    printf("Anticipated Profit(loss): $%#7.2f\n", pft(&first));

if(rflag)
    printf("Return on Funds Employed: %#3.2f%%\n", rfe(&first));
}

/* End of Main Module -- restate.c */

/* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{

```

(continued on next page)

```
        return(ps->i/12 * ps->t * ps->dp);
    }

        /* Profit -- pft.c */

#include "recdef.h"

float
pft(ps)
struct rec *ps;
{
    return(ps->spx - ps->ppx + ps->c);
}

        /* Return on Funds Employed -- rfe.c */

#include "recdef.h"

float
rfe(ps)
struct rec *ps;
{
    return(100 * (ps->spx - ps->c) / ps->spx);
}

        /* Header File -- recdef.h */

struct rec {
    char pname[25];
    float ppx;
    float dp;
    float i;
    float c;
    float t;
    float spx;
};
```

cflow

cflow produces a chart of the external references in C, **yacc**, **lex**, and assembly language files. Using the modules of our sample program, the command

```
cflow restate.c oppty.c pft.c rfe.c
```

produces the output shown in Figure 2-16.

```
1      main: int(), <restate.c 11>
2          fprintf: <>
3          exit: <>
4          getopt: <>
5          fopen: <>
6          fscanf: <>
7          printf: <>
8      oppty: float(), <oppty.c 7>
9      pft: float(), <pft.c 7>
10     rfe: float(), <rfe.c 8>
```

Figure 2-16: **cflow** Output, No Options

The `-r` option looks at the caller: callee relationship from the other side. It produces the output shown in Figure 2-17.

```
1      exit: <>
2          main : <>
3      fopen: <>
4          main : 2
5      fprintf: <>
6          main : 2
7      fscanf: <>
8          main : 2
9      getopt: <>
10         main : 2
11     main: int(), <restate.c 11>
12     oppty: float(), <oppty.c 7>
13         main : 2
14     pft: float(), <pft.c 7>
15         main : 2
16     printf: <>
17         main : 2
18     rfe: float(), <rfe.c 8>
19         main : 2
```

Figure 2-17: **cf**low Output, Using `-r` Option

The `-ix` option causes external and static data symbols to be included. Our sample program has only one such symbol, `opterr`. The output is shown in Figure 2-18.

```
1      main: int(), <restate.c 11>
2          fprintf: <>
3          exit: <>
4          opterr: <>
5          getopt: <>
6          fopen: <>
7          fscanf: <>
8          printf: <>
9          oppty: float(), <oppty.c 7>
10         pft: float(), <pft.c 7>
11         rfe: float(), <rfe.c 8>
```

Figure 2-18: `cflow` Output, Using `-ix` Option

Combining the `-r` and the `-ix` options produces the output shown in Figure 2-19.

```
1      exit: <>
2          main : <>
3      fopen: <>
4          main : 2
5      fprintf: <>
6          main : 2
7      fscanf: <>
8          main : 2
9      getopt: <>
10         main : 2
11     main: int(), <restate.c 11>
12     oppty: float(), <oppty.c 7>
13         main : 2
14     opterr: <>
15         main : 2
16     pft: float(), <pft.c 7>
17         main : 2
18     printf: <>
19         main : 2
20     rfe: float(), <rfe.c 8>
21         main : 2
```

Figure 2-19: `cflow` Output, Using `-r` and `-ix` Options

`ctrace`

`ctrace` lets you follow the execution of a C program statement by statement. `ctrace` takes a `.c` file as input and inserts statements in the source code to print out variables as each program statement is executed. You must direct the output of this process to a temporary `.c` file. The temporary file is then used as input to `cc`. When the resulting `a.out` file is executed it produces output that can tell you a lot about what is going on in your program.

Options give you the ability to limit the number of times through loops. You can also include functions in your source file that turn the trace off and on so you can limit the output to portions of the program that are of particular interest.

ctrace accepts only one source code file as input. To use our sample program to illustrate, it is necessary to execute the following four commands:

```
ctrace restate.c > ct.main.c  
ctrace oppty.c > ct.op.c  
ctrace pft.c > ct.p.c  
ctrace rfe.c > ct.r.c
```

The names of the output files are completely arbitrary. Use any names that are convenient for you. The names must end in **.c**, since the files are used as input to the C compilation system.

```
cc -o ct.run ct.main.c ct.op.c ct.p.c ct.r.c
```

Now the command

```
ct.run -opr
```

produces the output shown in Figure 2-20. The command above will cause the output to be directed to your terminal (**stdout**). It is probably a good idea to direct it to a file or to a printer so you can refer to it.

```
8 main(argc, argv)
23 if (argc < 2)
    /* argc == 2 */
30 opterr = FALSE;
    /* FALSE == 0 */
    /* opterr == 0 */
31 while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argv == 14679644 */
    /* argc == 2 */
    /* ch == 111 or 'o' */
32 {
33     switch(ch)
        /* ch == 111 or 'o' */
35     case 'o':
36         oflag = TRUE;
            /* TRUE == 1 */
            /* oflag == 1 */
37         break;
48 }
31 while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argv == */ 14679644 */
    /* argc == 2 */
    /* ch == 112 or 'p' */
32 {
33     switch(ch)
        /* ch == 112 or 'p' */
38     case 'p':
39         pflag = TRUE;
            /* TRUE == 1 */
            /* pflag == 1 */
40         break;
48 }
```

Figure 2-20: ctrace Output (sheet 1 of 3)

```

31 while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argv == 15679644 */
    /* argc == 2 */
    /* ch == 114 or 'r' */
32 {
33     switch(ch)
        /* ch == 114 or 'r' */
41     case 'r':
42         rflag = TRUE;
            /* TRUE == 1 */
            /* rflag == 1 */
43         break;
44     }
31 while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argv == 15679644 */
    /* argc == 2 */
    /* ch == -1 */
49 if ((fin = fopen("info","r")) == NULL)
    /* fin == 3149530 */
54 if (fscanf(fin, "%s%f%f%f%f%f",first.pname,&first.ppx,
    &first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
    /* fin == 3149530 */
    /* first.pname == 14679572 */
61 printf("Property: %s0,first.pname);
    /* first.pname == 14679572 or "Linden_Place" */ Property: Linden_Place

63 if(oflag)
    /* oflag == 1 or */
64     printf("Opportunity Cost: $%#5.2f0,oppty(&first));
5 oppty(ps)
8 return(ps->i/12 * ps->t * ps->dp);
    /* ps->dp == 1088765312 */ Opportunity Cost: $4476.87
    /* ps->t == 1076494336 */
    /* ps->i == 1069044203 */

```

Figure 2-20: ctrace Output (sheet 2 of 3)

```

66  if(pflag)
    /* pflag == 1 */
67      printf("Anticipated Profit(loss): $%#7.2f0,pft(&first));
5  pft(ps)
8  return(ps->spx - ps->ppx + ps->c);
    /* ps->c == 1087409536 */      Anticipated Profit(loss): $85950.00
    /* ps->spx == 1091649040 */
    /* ps->ppx == 1091178464 */

69  if(rflag)
    /* rflag == 1 */
70      printf("Return on Funds Employed: %#3.2f%%0,rfe(&first));
6  rfe(ps)
9  return(100 * (ps->spx - ps->c) / ps->spx);
    /* ps->spx == 1091649040 */
    /* ps->c == 1087409536 */ Return on Funds Employed: 94.00%

/* return */

```

Figure 2-20: **ctrace** Output (sheet 3 of 3)

Using a program that runs successfully is not the optimal way to demonstrate **ctrace**. It would be more helpful to have an error in the operation that could be detected by **ctrace**. It would seem that this utility might be most useful in cases where the program runs to completion, but the output is not as expected.

cxref

cxref analyzes a group of C source code files and builds a cross-reference table of the automatic, static, and global symbols in each file. The command

```
cxref -c -o cx.op restate.c oppty.c pft.c rfe.c
```

produces the output shown in Figure 2-21 in a file named, in this case, **cx.op**. The **-c** option causes the reports for the four **.c** files to be combined in one cross-reference file.

```

restate.c:

oppty.c:

pft.c:

rfe.c:

SYMBOL          FILE          FUNCTION      LINE

BUFSIZ          /usr/include/stdio.h  --            *9
EOF             /usr/include/stdio.h  --            49 *50
               restate.c          --            31
FALSE          /usr/include/stdio.h  --            *6 15 16 17 30
FILE           /usr/include/stdio.h  --            *29 73 74
               restate.c          main          12
L_ctermid      /usr/include/stdio.h  --            *80
L_cuserid      /usr/include/stdio.h  --            *81
L_tmpnam       /usr/include/stdio.h  --            *83
NULL           /usr/include/stdio.h  --            46 *47
               restate.c          --            49
P_tmpdir       /usr/include/stdio.h  --            *82
TRUE           restate.c            --            *5 36 39 42
_IOEOF         /usr/include/stdio.h  --            *41
_IOERR         /usr/include/stdio.h  --            *42
_IOFBF        /usr/include/stdio.h  --            *36
_IOLBF        /usr/include/stdio.h  --            *43
_IOMYBUF      /usr/include/stdio.h  --            *40
_IONBF        /usr/include/stdio.h  --            *39
_IOREAD       /usr/include/stdio.h  --            *37
_IORW         /usr/include/stdio.h  --            *44
_IOWRT        /usr/include/stdio.h  --            *38
_NFILE        /usr/include/stdio.h  --            2 *3 73
_SBFSIZ       /usr/include/stdio.h  --            *16
    
```

 Figure 2-21: **cxref** Output, Using **-c** Option (sheet 1 of 5)

Analysis/Debugging

SYMBOL	FILE	FUNCTION	LINE
_base	/usr/include/stdio.h	--	*26
_bufend()	/usr/include/stdio.h	--	*57
_bufendtab	/usr/include/stdio.h	--	*78
_bufsiz()	/usr/include/stdio.h	--	*58
_cnt	/usr/include/stdio.h	--	*20
_file	/usr/include/stdio.h	--	*28
_flag	/usr/include/stdio.h	--	*27
_iob	/usr/include/stdio.h	--	*73
_ptr	restate.c	main	25 26 45 51 57
argc	/usr/include/stdio.h	--	*21
	restate.c	--	8
argv	restate.c	main	*9 23 31
	restate.c	--	8
c	restate.c	main	*10 25 26 31 45 51 57
	./recdef.h	--	*6
	pft.c	pft	8
	restate.c	main	55
	rfe.c	rfe	9
ch	restate.c	main	*18 31 33
clearerr()	/usr/include/stdio.h	--	*67
ctermid()	/usr/include/stdio.h	--	*77
cuserid()	/usr/include/stdio.h	--	*77
dp	./recdef.h	--	---*4
	oppty.c	oppty	8
	restate.c	main	55
exit()	restate.c	main	*13 27 46 52 58
fdopen()	/usr/include/stdio.h	--	*74

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 2 of 5)

SYMBOL	FILE	FUNCTION	LINE
feof()	/usr/include/stdio.h	--	*68
ferror()	/usr/include/stdio.h	--	*69
fgets()	/usr/include/stdio.h	--	*77
fileno()	/usr/include/stdio.h	--	*70
fin	restate.c	main	*12 49 54
first	restate.c	main	*19 54 55 61 64 67 70
fopen()	/usr/include/stdio.h	--	*74
	restate.c	main	12 49
fprintf	restate.c	main	25 26 45 51 57
freopen()	/usr/include/stdio.h	--	*74
fscanf	restate.c	main	54
ftell()	/usr/include/stdio.h	--	*75
getc()	/usr/include/stdio.h	--	*61
getchar()	/usr/include/stdio.h	--	*65
getopt()	restate.c	main	*14 31
gets()	/usr/include/stdio.h	--	*77
i	./recdef.h	--	*5
	oppty.c	oppty	8
	restate.c	main	55
lint	/usr/include/stdio.h	--	60
main()	restate.c	--	*8

 Figure 2-21: **cxref** Output, Using **-c** Option (sheet 3 of 5)

SYMBOL	FILE	FUNCTION	LINE
oflag	restate.c	main	*15 36 63
oppty()	oppty.c	--	*5
	restate.c	main	*21 64
opterr	restate.c	main	*20 30
p	/usr/include/stdio.h	--	*57 *58 *61 62
			*62 63 64 67 *67 68 *68 69 *69 70 *70
pdpl1	/usr/include/stdio.h	--	11
pflag	restate.c	main	*16 39 66
	pft.c	--	*5
	restate.c	main	*21 67
pname	./recdef.h	--	*2
	restate.c	main	54 61
popen()	/usr/include/stdio.h	--	*74
ppx	./recdef.h	--	*3
	pft.c	pft	8
	restate.c	main	54
printf	restate.c	main	61 64 67 70
ps	oppty.c	--	5
	oppty.c	oppty	*6 8
	pft.c	--	5
	pft.c	pft	*6 8
	rfe.c	--	6
	rfe.c	rfe	*7 9
putc()	/usr/include/stdio.h	--	*62
putchar()	/usr/include/stdio.h	--	*66
rec	./recdef.h	--	*1
	oppty.c	oppty	6
	pft.c	pft	6
	restate.c	main	19
	rfe.c	rfe	7

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 4 of 5)

SYMBOL	FILE	FUNCTION	LINE
rewind()	/usr/include/stdio.h	--	*76
rfe()	restate.c	main	*21 70
	rfe.c	--	*6
rflag	restate.c	main	*17 42 69
setbuf()	/usr/include/stdio.h	--	*76
spx	./recdef.h	--	*8
	pft.c	pft	8
	restate.c	main	55
	rfe.c	rfe	9
stderr	/usr/include/stdio.h	--	*55
	restate.c	--	25 26 45 51 57
stdin	/usr/include/stdio.h	--	*53
stdout	/usr/include/stdio.h	--	*54
t	./recdef.h	--	*7
	oppty.c	oppty	8
	restate.c	main	55
tempnam()	/usr/include/stdio.h	--	*77
tmpfile()	/usr/include/stdio.h	--	*74
tmpnam()	/usr/include/stdio.h	--	*77
u370	/usr/include/stdio.h	--	5
u3b	/usr/include/stdio.h	--	8 19
u3b5	/usr/include/stdio.h	--	8 19
vax	/usr/include/stdio.h	--	8 19
x	/usr/include/stdio.h	--	*62 63 64 66 *66

 Figure 2-21: **cxref** Output, Using **-c** Option (sheet 5 of 5)

lint

lint looks for features in a C program that are apt to cause execution errors, that are wasteful of resources, or that create problems of portability.

The command

```
lint restate.c oppty.c pft.c rfe.c
```

produces the output shown in Figure 2-22.

```
restate.c:
restate.c
=====
(71) warning: main() returns random value to invocation environment
oppty.c:
pft.c:
rfe.c:

=====
function returns value which is always ignored
printf
```

Figure 2-22: **lint** Output

lint has options that will produce additional information. Check the *System V Reference Manual*. The error messages give you the line numbers of some items you may want to review.

size

size produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Here are the results of one invocation of the **size** command with our object file as an argument. The output from the **size** routine depends on processor type and thereby on the TARGETMC:

TARGETMC = 68020:

10212(.text) + 3396(.data) + 5256(.bss) + 0(.stack) = 18864

TARGETMC = R3KMI:

14736(.text) + 32(.init) + 368(.rdata) + 2928(.data) +
144(.lit8) + 224(.sdata) + 48(.sbss) + 2960(.bss) = 21440

TARGETMC = R3KMO:

15840(.text) + 32(.init) + 368(.rdata) + 2704(.data) +
144(.lit8) + 224(.sdata) + 48(.sbss) + 2960(.bss) = 22320

Don't confuse this number with the number of characters in the object file that appears when you do an **ls -l** command. That figure includes the symbol table and other header information that is not used at run time.

strip

strip removes the symbol and line number information from a common object file. When you issue this command the number of characters shown by the **ls -l** command approaches the figure shown by the **size** command, but still includes some header information that is not counted as part of the .text, .data, or .bss section. After the **strip** command has been executed, it is no longer possible to use the file with the **sdb** command.

sdb/dbx

sdb stands for Symbolic Debugger, which means you can use the symbolic names in your program to pinpoint where a problem has occurred. You can use **sdb** to debug C, or PASCAL programs. There are two basic ways to use **sdb**: by running your program under control of **sdb**, or by using **sdb** to rummage through a core image file left by a program that failed. The first way lets you see what the program is doing up to the point at which it fails (or to skip around the failure point and proceed with the run). The second method lets you check the status at the moment of failure, which may or may not disclose the reason the program failed.

Chapter 14 of **sdb** describes the interactive commands you can use to work your way through your program. For the time being we want to tell you just a couple of key things you need to do when using it.

1. Compile your program(s) with the **-g** option, which causes additional information to be generated for use by **sdb**.
2. Run your program under **sdb** with the command:

```
sdb myprog - srcdir
```

where **myprog** is the name of your executable file (**a.out** is the default), and **srcdir** is an optional list of the directories where source code for your modules may be found. The dash between the two arguments keeps **sdb** from looking for a core image file.

NOTE

sdb is for 68xxx based programs; while **dbx** is used for the R3000 based programs.

Program Organizing Utilities

The following three utilities are helpful in keeping your programming work organized effectively.

The **make** Command

When you have a program that is made up of more than one module of code you begin to run into problems of keeping track of which modules are up to date and which need to be recompiled when changes are made in another module. The **make** command is used to ensure that dependencies between modules are recorded so that changes in one module results in the re-compilation of dependent programs. Even control of a program as simple as the one shown in Figure 2-15 is made easier through the use of **make**.

The **make** utility requires a description file that you create with an editor. The description file (also referred to by its default name: **makefile**) contains the information used by **make** to keep a target file current. The target file is typically an executable program. A description file contains three types of information:

- 1) dependency information tells the **make** utility the relationship between the modules that comprise the target program.
- 2) executable commands needed to generate the target program. **make** uses the dependency information to determine which executable commands should be passed to the shell for execution.

Program Organizing Utilities

- 3) macro definitions provide a shorthand notation within the description file to make maintenance easier. Macro definitions can be overridden by information from the command line when the **make** command is entered.

The **make** command works by checking the "last changed" time of the modules named in the description file. When **make** finds a component that has been changed more recently than modules that depend on it, the specified commands (usually compilations) are passed to the shell for execution.

The **make** command takes three kinds of arguments: options, macro definitions, and target filenames. If no description filename is given as an option on the command line, **make** searches the current directory for a file named **makefile** or **Makefile**. Figure 2-23 shows a **makefile** for our sample program.

```
OBJECTS = restate.o oppty.o pft.o rfe.o
all: restate
restate: $(OBJECTS)
        $(CC) $(CFLAGS) $(LDFLAGS) $(OBJECTS) -o restate
$(OBJECTS): ./reconf.h
clean:
        rm -f $(OBJECTS)
clobber: clean
        rm -f restate
```

Figure 2-23: **make** Description File

The following things are worth noticing in this description file:

- It identifies the target, **restate**, as being dependent on the four object modules. Each of the object modules in turn is defined as being dependent on the header file, **recdef.h**, and by default, on its corresponding source file.
- A macro, **OBJECTS**, is defined as a convenient shorthand for referring to all of the component modules.

Whenever testing or debugging results in a change to one of the components of **restate**, for example, a command such as the following should be entered:

```
make CFLAGS = -g restate
```

This has been a very brief overview of the **make** utility. There is more on **make** in Chapter 3, and a detailed description of **make** can be found in Chapter 12.

The Archive

The most common use of an archive file, although not the only one, is to hold object modules that make up a library. The library can be named on the link editor command line (or with a link editor option on the **cc** command line). This causes the link editor to search the symbol table of the archive file when attempting to resolve references.

The **ar** command is used to create an archive file, to manipulate its contents and to maintain its symbol table. The structure of the **ar** command is a little different from the normal UNIX system arrangement of command line options. When you enter the **ar** command you include a one-character key from the set **drqtpmx** that defines the type of action you intend. The key may be combined with one or more additional characters from the set **vuai bcl s** that modify the way the requested operation is performed. The makeup of the command line is

Program Organizing Utilities

```
ar -key [posname] afile [name]...
```

where *posname* is the name of a member of the archive and may be used with some optional key characters to make sure that the files in your archive are in a particular order. The *afile* argument is the name of your archive file. By convention, the suffix **.a** is used to indicate the named file is an archive file. (**libc.a**, for example, is the archive file that contains many of the object files of the standard C subroutines.) One or more *names* may be furnished. These identify files that are subjected to the action specified in the *key*.

We can make an archive file to contain the modules used in our sample program, **restate**. The command to do this is

```
ar -rv rste.a restate.o oppty.o pft.o rfe.o
```

If these are the only **.o** files in the current directory, you can use shell metacharacters as follows:

```
ar -rv rste.a *.o
```

Either command will produce this feedback:

```
a - restate.o
a - oppty.o
a - pft.o
a - rfe.o
ar: creating rste.a
```

The **nm** command is used to get a variety of information from the symbol table of common object files. The object files can be, but don't have to be, in an archive file. Figure 2-24 shows the output of this command when executed with the **-f** (for full) option on the archive we just created. The object files were compiled with the **-g** option.

Symbols from rste.a[restate.o]

Name	Val	Class	Type	Size	Line	Sect
restate.c		file				
.0fake		strtag	struct	14		
_cnt	0	strmem	int			(ABS)
_ptr	4	strmem	*Uchar			(ABS)
_base	8	strmem	*Uchar			(ABS)
_flag	12	strmem	char			(ABS)
_file	13	strmem	Uchar			(ABS)
.eos		endstr		14		(ABS)
rec		strtag	struct	50		
pname	0	strmem	char[25]	25		(ABS)
ppx	26	strmem	float			(ABS)
dp	30	strmem	float			(ABS)
i	34	strmem	float			(ABS)
c	38	strmem	float			(ABS)
t	42	strmem	float			(ABS)
spx	46	strmem	float			(ABS)
.eos		endstr		50		(ABS)
main	0	extern	int()	504		.text
.bf	0	fcn			11	.text
argc	6	regprm	int			(ABS)
argv	13	regprm	**char			(ABS)
fin	12	reg	*struct-.0fake	14		(ABS)
oflag	5	reg	int			(ABS)
pflag	4	reg	int			(ABS)
rflag	3	reg	int			(ABS)
ch	2	reg	int			(ABS)

Figure 2-24: nm Output, with -f Option (sheet 1 of 5)

Program Organizing Utilities

Symbols from `rste.a[restate.o]`

Name	Val	Class	Type	Size	Line	Sect
<code>first</code>	52	auto	struct-rec	50		(ABS)
<code>.ef</code>	496	fcn			61	.text
<code>FILE</code>		typedef	struct-.0fake	16		
<code>_iob</code>	0	extern				
<code>fprintf</code>	0	extern				
<code>exit</code>	0	extern				
<code>opterr</code>	0	extern				
<code>getopt</code>	0	extern				
<code>fopen</code>	0	extern				
<code>fscanf</code>	0	extern				
<code>printf</code>	0	extern				
<code>oppty</code>	0	extern	float()			
<code>pft</code>	0	extern	float()			
<code>rfe</code>	0	extern	float()			

Figure 2-24: nm Output, with `-f` Option (sheet 2 of 5)

Symbols from rste.a[oppty.o]

Name	Val	Class	Type	Size	Line	Sect
oppty.c		file				
rec		strtag	struct	50		
pname	0	strmem	char[25]	25		
ppx	26	strmem	float			
dp	30	strmem	float			
i	34	strmem	float			
c	38	strmem	float			
t	42	strmem	float			
spx	46	strmem	float			
.eos		endstr		50		
oppty	0	extern	float()	38		.text
.bf	10	fcn			7	.text
ps	8	regprm	*struct-rec	50		
.ef	34	fcn			3	.text

Figure 2-24: nm Output, with -f Option (sheet 3 of 5)

Program Organizing Utilities

Symbols from rste.a[pft.o]

Name	Val	Class	Type	Size	Line	Sect
pft.c		file				
rec		strtag	struct	50		
pname	0	strmem	char[25]	25		
ppx	26	strmem	float			
dp	30	strmem	float			
i	34	strmem	float			
c	38	strmem	float			
t	42	strmem	float			
spx	46	strmem	float			
.eos		endstr		50		
pft	0	extern	float()	60		.text
.bf	10	fcn			7	.text
ps	0	argm't	*struct-rec	50		
.ef	58	fcn			3	.text

Figure 2-24: nm Output, with **-f** Option (sheet 4 of 5)

Symbols from rste.a[rfe.o]

Name	Val	Class	Type	Size	Line	Sect
rfe.c		file				
rec		strtag	struct	50		
pname	0	strmem	char[25]	25		
ppx	26	strmem	float			
dp	30	strmem	float			
i	34	strmem	float			
c	38	strmem	float			
t	42	strmem	float			
spx	46	strmem	float			
.eos		endstr		50		
rfe	0	extern	float()	68		.text
.bf	10	fcn			8	.text
ps	0	argm't	*struct-rec	50		
.ef	64	fcn			3	.text

Figure 2-24: nm Output, with -f Option (sheet 5 of 5)

For **nm** to work on an archive file all of the contents of the archive have to be object modules. If you have stored other things in the archive, you will get the message:

```
nm: rste.a bad magic
```

when you try to execute the command.

Use of SCCS by Single-User Programmers

The UNIX system Source Code Control System (SCCS) is a set of programs designed to keep track of different versions of programs. When a program has been placed under control of SCCS, only a single copy of any one version of the code can be retrieved for editing at a given time. When program code is changed and the program returned to SCCS, only the changes are recorded. Each version of the code is

Program Organizing Utilities

identified by its SID, or **SCCS ID**entifying number. By specifying the SID when the code is extracted from the SCCS file, it is possible to return to an earlier version. If an early version is extracted with the intent of editing it and returning it to SCCS, a new branch of the development tree is started. The set of programs that make up SCCS appear as UNIX system commands. The commands are:

- admin**
- get**
- delta**
- prs**
- rmdel**
- cdc**
- what**
- sccsdiff**
- comb**
- val**

It is most common to think of SCCS as a tool for project control of large programming projects. It is, however, entirely possible for any individual user of the UNIX system to set up a private SCCS system. Chapter 13 is an SCCS user's guide.

Chapter 3: Application Programming

	Page
Introduction.....	3- 1
Application Programming.....	3- 3
Numbers.....	3- 3
Portability.....	3- 4
Documentation.....	3- 4
Project Management.....	3- 5
Language Selection.....	3- 7
Influences.....	3- 7
Special Purpose Languages.....	3- 8
What awk Is Like.....	3- 8
How awk Is Used.....	3- 9
Where to Find More Information.....	3-10
What lex and yacc Are Like.....	3-10
How lex Is Used.....	3-11
Where to Find More Information.....	3-13
How yacc Is Used.....	3-13
Where to Find More Information.....	3-15
Advanced Programming Tools.....	3-17
Memory Management.....	3-17
File and Record Locking.....	3-18
How File and Record Locking Works.....	3-19
lockf	3-21
Where to Find More Information.....	3-21
Interprocess Communications.....	3-22

Table of Contents

	Page
IPC get Calls	3-23
IPC ctl Calls	3-23
IPC op Calls	3-23
Where to Find More Information	3-24
Programming Terminal Screens	3-24
curses	3-25
Where to Find More Information	3-25
Programming Support Tools	3-27
Link Edit Command Language	3-27
Where to Find More Information	3-28
Common Object File Format	3-28
Where to Find More Information	3-29
Libraries	3-29
The Object File Library	3-30
Common Object File Interface Macros (ldfcn.h)	3-32
The Math Library	3-33
Trigonometric Functions	3-34
Bessel Functions	3-34
Hyperbolic Functions	3-35
Miscellaneous Functions	3-35
Symbolic Debugger	3-36
Where to Find More Information	3-37
lint as a Portability Tool	3-37
Where to Find More Information	3-38
Project Control Tools	3-39
make	3-39
Where to Find More Information	3-40

Table of Contents

	Page
SCCS.....	3-40
Where to Find More Information.....	3-42
liber , A Library System.....	3-43

Table of Contents

This page is intentionally left blank

Introduction

This chapter deals with programming where the objective is to produce sets of programs (applications) that will run on a UNIX system computer.

The chapter begins with a discussion of how the ground rules change as you move up the scale from writing programs that are essentially for your own private use (we have called this single-user programming), to working as a member of a programming team developing an application that is to be turned over to others to use.

There is a section on how the criteria for selecting appropriate programming languages may be influenced by the requirements of the application.

The next three sections of the chapter deal with a number of loosely-related topics that are of importance to programmers working in the application development environment. Most of these mirror topics that were discussed in Chapter 2, Programming Basics, but here we try to point out aspects of the subject that are particularly pertinent to application programming. They are covered under the following headings:

- | | |
|-----------------------|--|
| Advanced Programming | deals with such topics as File and Record Locking, Interprocess Communication, and programming terminal screens. |
| Support Tools | covers the Common Object File Format, link editor directives, shared libraries, sdb/dbx , and lint . |
| Project Control Tools | includes some discussion of make and SCCS . |

The chapter concludes with a description of a sample application called **liber** that uses several of the components described in earlier portions of the chapter.

Introduction

This page is intentionally left blank

Application Programming

The characteristics of the application programming environment that make it different from single-user programming have at their base the need for interaction and for sharing of information.

Numbers

Perhaps the most obvious difference between application programming and single-user programming is in the quantities of the components. Not only are applications generally developed by teams of programmers, but the number of separate modules of code can grow into the hundreds on even a fairly simple application.

When more than one programmer works on a project, there is a need to share such information as:

- the operation of each function
- the number, identity and type of arguments expected by a function
- if pointers are passed to a function, are the objects being pointed to modified by the called function, and what is the lifetime of the pointed-to object
- the data type returned by a function

In an application, there is an odds-on possibility that the same function can be used in many different programs, by many different programmers. The object code needs to be kept in a library accessible to anyone on the project who needs it.

Portability

When you are working on a program to be used on a single model of a computer, your concerns about portability are minimal. In application development, on the other hand, a desirable objective often is to produce code that will run on many different UNIX system computers. Some of the things that affect portability will be touched on later in this chapter.

Documentation

A single-user program has modest needs for documentation. There should be enough to remind the program's creator how to use it, and what the intent was in portions of the code.

On an application development project there is a significant need for two types of internal documentation:

- comments throughout the source code that enable successor programmers to understand easily what is happening in the code. Applications can be expected to have a useful life of 5 or more years, and frequently need to be modified during that time. It is not realistic to expect that the same person who wrote the program will always be available to make modifications. Even if that does happen the comments will make the maintenance job a lot easier.
- hard-copy descriptions of functions should be available to all members of an application development team. Without them it is difficult to keep track of available modules, which can result in the same function being written over again.

Unless end-users have clear, readily-available instructions in how to install and use an application they either will not do it at all (if that is an option), or do it improperly.

The microcomputer software industry has become ever more keenly aware of the importance of good end-user documentation. There are cases on record where the success of a software package has been attributed in large part to the fact that it had exceptionally good documentation. There are also cases where a pretty good piece of software was not widely used due to the inaccessibility of its manuals. There appears to be no truth to the rumor that in one or two cases, end-users have thrown the software away and just read the manual.

Project Management

Without effective project management, an application development project is in trouble. This subject will not be dealt with in this guide, except to mention the following three things that are vital functions of project management:

- tracking dependencies between modules of code
- dealing with change requests in a controlled way
- seeing that milestone dates are met

This page is intentionally left blank

Language Selection

In this section we talk about some of the considerations that influence the selection of programming languages, and describe two of the special purpose languages that are part of the UNIX system environment.

Influences

In single-user programming the choice of language is often a matter of personal preference; a language is chosen because it is the one the programmer feels most comfortable with.

An additional set of considerations comes into play when making the same decision for an application development project.

- 131
- Q:** Is there an existing standard within the organization that should be observed?
- A:** A firm may decide to emphasize one language because a good supply of programmers is available who are familiar with it.
- Q:** Does one language have better facilities for handling the particular algorithm?
- A:** One would like to see all language selection based on such objective criteria, but it is often necessary to balance this against the skills of the organization.
- Q:** Is there an inherent compatibility between the language and the UNIX operating system?
- A:** This is sometimes the impetus behind selecting C for programs destined for a UNIX system machine.

Language Selection

- Q:** Are there existing tools that can be used?
- A:** If parsing of input lines is an important phase of the application, perhaps a parser generator such as **yacc** should be employed to develop what the application needs.
- Q:** Does the application integrate other software into the whole package?
- A:** If, for example, a package is to be built around an existing data base management system, there may be constraints on the variety of languages the data base management system can accommodate.

Special Purpose Languages

The UNIX system contains a number of tools that can be included in the category of special purpose languages. Three that are especially interesting are **awk**, **lex**, and **yacc**.

What **awk** Is Like

The **awk** utility scans an ASCII input file record by record, looking for matches to specific patterns. When a match is found, an action is taken. Patterns and their accompanying actions are contained in a specification file referred to as the program. The program can be made up of a number of statements. However, since each statement has the potential for causing a complex action, most **awk** programs consist of only a few. The set of statements may include definitions of the pattern that separates one record from another (a newline character, for example), and what separates one field of a record from the next (white space, for example). It may also include actions to be performed before the first record of the input file is read, and other actions to be performed after the final record has been read. All statements in between are evaluated in order for each record in the input file.

To paraphrase the action of a simple **awk** program, it would go something like this:

Look through the input file.
Every time you see this specific pattern, do this action.

A more complex **awk** program might be paraphrased like this:

First do some initialization.
Then, look through the input file.
Every time you see this specific pattern, do this action.
Every time you see this other pattern, do another action.
After all the records have been read, do these final things.

The directions for finding the patterns and for describing the actions can get pretty complicated, but the essential idea is as simple as the two sets of statements above.

One of the strong points of **awk** is that once you are familiar with the language syntax, programs can be written very quickly. They don't always run very fast, however, so they are seldom appropriate if you want to run the same program repeatedly on a large quantities of records. In such a case, it is likely to be better to translate the program to a compiled language.

How **awk** Is Used

One typical use of **awk** would be to extract information from a file and print it out in a report. Another might be to pull fields from records in an input file, arrange them in a different order and pass the resulting rearranged data to a function that adds records to your data base. There is an example of a use of **awk** in the sample application at the end of this chapter.

Where to Find More Information

The manual page for **awk** is in Section (1) of the *System V Reference Manual*. Chapter 4 contains a description of the **awk** syntax and a number of examples showing ways in which **awk** may be used.

What **lex** and **yacc** Are Like

lex and **yacc** are often mentioned in the same breath because they perform complementary parts of what can be viewed as a single task: making sense out of input. The two utilities also share the common characteristic of producing source code for C language subroutines from specifications that appear on the surface to be quite similar.

Recognizing input is a recurring problem in programming. Input can be from various sources. In a language compiler, for example, the input is normally contained in a file of source language statements. The UNIX system shell language most often receives its input from a person keying in commands from a terminal. Frequently, information coming out of one program is fed into another where it must be evaluated.

The process of input recognition can be subdivided into two tasks: lexical analysis and parsing, and that's where **lex** and **yacc** come in. In both utilities, the specifications cause the generation of C language subroutines that deal with streams of characters; **lex** generates subroutines that do lexical analysis while **yacc** generates subroutines that do parsing.

To describe those two tasks in dictionary terms:

- Lexical analysis has to do with identifying the words or vocabulary of a language as distinguished from its grammar or structure.
- Parsing is the act of describing units of the language grammatically. Students in elementary school are often taught to do this with sentence diagrams.

Of course, the important thing to remember here is that in each case the rules for our lexical analysis or parsing are those we set down ourselves in the **lex** or **yacc** specifications. Because of this, the dividing line between lexical analysis and parsing sometimes becomes fuzzy.

The fact that **lex** and **yacc** produce C language source code means that these parts of what may be a large programming project can be separately maintained. The generated source code is processed by the C compiler to produce an object file. The object file can be link edited with others to produce programs that then perform whatever process follows from the recognition of the input.

How **lex** Is Used

A **lex** subroutine scans a stream of input characters and waves a flag each time it identifies something that matches one or another of its rules. The waved flag is referred to as a token. The rules are stated in a format that closely resembles the one used by the UNIX system text editor for regular expressions. For example,

```
[ \t]+
```

describes a rule that recognizes a string of one or more blanks or tabs (without mentioning any action to be taken). A more complete statement of that rule might have this notation:

```
[ \t]+ ;
```

which, in effect, says to ignore white space. It carries this meaning because no action is specified when a string of one or more blanks or tabs is recognized. The semicolon marks the end of the statement. Another rule, one that does take some action, could be stated like this:

```
[0-9]+ {  
    i = atoi(yytext);  
    return(NBR);  
}
```

Language Selection

This rule depends on several things:

- **NBR** must have been defined as a token in an earlier part of the **lex** source code called the declaration section. (It may be in a header file which is **#include**'d in the declaration section.)
- **i** is declared as an **extern int** in the declaration section.
- It is a characteristic of **lex** that things it finds are made available in a character string called **ytext**.
- Actions can make use of standard C syntax. Here, the standard C subroutine, **atoi**, is used to convert the string to an integer.

What this rule boils down to is **lex** saying, "Hey, I found the kind of token we call **NBR**, and its value is now in **i**."

To review the steps of the process:

1. The **lex** specification statements are processed by the **lex** utility to produce a file called **lex.yy.c**. (This is the standard name for a file generated by **lex**, just as **a.out** is the standard name for the executable file generated by the link editor.)
2. **lex.yy.c** is transformed by the C compiler (with a **-c** option) into an object file called **lex.yy.o** that contains a subroutine called **yylex()**.
3. **lex.yy.o** is link edited with other subroutines. Presumably one of those subroutines will call **yylex()** with a statement such as:

```
while((token = yylex()) != 0)
```

and other subroutines (or even **main**) will deal with what comes back.

Where to Find More Information

The manual page for **lex** is in Section (1) of the *System V Reference Manual*. A tutorial on **lex** is contained in Chapter 5.

How yacc Is Used

yacc subroutines are produced by pretty much the same series of steps as **lex**:

1. The **yacc** specification is processed by the **yacc** utility to produce a file called **y.tab.c**.
2. **y.tab.c** is compiled by the C compiler producing an object file, **y.tab.o**, that contains the subroutine **yyparse()**. A significant difference is that **yyparse()** calls a subroutine called **yylex()** to perform lexical analysis.
3. The object file **y.tab.o** may be link edited with other subroutines, one of which will be called **yylex()**.

There are two things worth noting about this sequence:

1. The parser generated by the **yacc** specifications calls a lexical analyzer to scan the input stream and return tokens.
2. While the lexical analyzer is called by the same name as one produced by **lex**, it does not have to be the product of a **lex** specification. It can be any subroutine that does the lexical analysis.

What really differentiates these two utilities is the format for their rules. As noted above, **lex** rules are regular expressions like those used by UNIX system editors. **yacc** rules are chains of definitions and alternative definitions, written in Backus-Naur form, accompanied by actions. The rules may refer to other rules defined further down the specification. Actions are sequences of C language statements enclosed in braces. They frequently contain numbered variables that enable you to reference values associated with parts of the rules. An example might make that easier to understand:

Language Selection

```

%token  NUMBER
%%
expr   : numb                { $$ = $1; }
       | expr '+' expr      { $$ = $1 + $3; }
       | expr '-' expr      { $$ = $1 - $3; }
       | expr '*' expr      { $$ = $1 * $3; }
       | expr '/' expr      { $$ = $1 / $3; }
       | '(' expr ')'        { $$ = $2; }
       ;
numb   : NUMBER              { $$ = $1; }
       ;

```

This fragment of a **yacc** specification shows:

- **NUMBER** identified as a token in the declaration section
- the start of the rules section indicated by the pair of percent signs
- a number of alternate definitions for *expr* separated by the | sign and terminated by the semicolon
- actions to be taken when a rule is matched
- within actions, numbered variables used to represent components of the rule:
 - $$$$ means the value to be returned as the value of the whole rule
 - $$n$ means the value associated with the *n*th component of the rule, counting from the left
- *numb* defined as meaning the token **NUMBER**. This is a trivial example that illustrates that one rule can be referenced within another, as well as within itself.

As with **lex**, the compiled **yacc** object file will generally be link edited with other subroutines that handle processing that takes place after the parsing—or even ahead of it.

Where to Find More Information

The manual page for **yacc** is in Section (1) of the *System V Reference Manual*. A detailed description of **yacc** may be found in Chapter 6 of this guide.

Language Selection

This page is intentionally left blank

Advanced Programming Tools

In Chapter 2 we described the use of such basic elements of programming in the UNIX system environment as the standard I/O library, header files, system calls and subroutines. In this section we introduce tools that are more apt to be used by members of an application development team than by a single-user programmer. The section contains material on the following topics:

- memory management
- file and record locking
- interprocess communication
- programming terminal screens

Memory Management

There are situations where a program needs to ask the operating system for blocks of memory. It may be, for example, that a number of records have been extracted from a data base and need to be held for some further processing. Rather than writing them out to a file on secondary storage and then reading them back in again, it is likely to be a great deal more efficient to hold them in memory for the duration of the process. (This is not to ignore the possibility that portions of memory may be paged out before the program is finished; but such an occurrence is not pertinent to this discussion.) There are two C language subroutines available for acquiring blocks of memory and they are both called **malloc**. One of them is **malloc(3C)**, the other is **malloc(3X)**. Each has several related commands that do specialized tasks in the same area. They are:

- **free** — to inform the system that space is being relinquished
- **realloc** — to change the size and possibly move the block

Advanced Programming Tools

- **calloc** — to allocate space for an array and initialize it to zeros

In addition, **malloc(3X)** has a function, **mallopt**, that provides for control over the space allocation algorithm, and a structure, **malinfo**, from which the program can get information about the usage of the allocated space.

malloc(3X) runs faster than the other version. It is loaded by specifying

– **lmalloc**

on the **cc(1)** or **ld(1)** command line to direct the link editor to the proper library. When you use **malloc(3X)** your program should contain the statement

```
#include <malloc.h>
```

where the values for **mallopt** options are defined.

See the *System V Reference Manual* for the formal definitions of the two **mallocs**.

File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies the whole file is affected; record locking means that only a specified portion of the file is locked. (Remember, in the UNIX system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: read locks and write locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it, that is, changing any of the data. If a process places a write lock on a file, no other processes can read or write in the file until the lock is removed. Write locks are also known as exclusive locks. The term shared lock is sometimes applied to read locks.

Mandatory locking means that the system prevents other processes from reading and writing a file or record if the action is incompatible with the lock. If for instance a process sets a write-lock on a file no other process will be able to read or write that file before the lock is removed.

Advisory locking is sort of a gentleman agreement. Even though a lock is set on a file, other processes may read and write that file. This mechanism require that the programs involved act gentle. Before a read or write is performed the corresponding lock must be effective. If the lock request is rejected the file has been locked by another process. The gentleman agreement requires you not to perform the reading or writing procedure before the lock request is accepted. After performing the read/write, the lock should be removed to allow access to this file.

How File and Record Locking Works

The system call for file and record locking is **fcntl(2)**. Programs should include the line

```
#include <fcntl.h>
```

to bring in the header file shown in Figure 3-1.

Advanced Programming Tools

```

/* Flag values accessible to open(2) and fcntl(2) */
/* (The first three can only be set by open) */
#define O_RDONLY    0
#define O_WRONLY    1
#define O_RDWR     2
#define O_NDELAY    04 /* Non-blocking I/O */
#define O_APPEND    010 /* append (writes guaranteed at the end) */
#define O_SYNC      020 /* synchronous write option */

/* Flag values accessible only to open(2) */
#define O_CREAT     00400 /* open with file create (uses third open arg)*/
#define O_TRUNC     01000 /* open with truncation */
#define O_EXCL      02000 /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD     0 /* Duplicate fildes */
#define F_GETFD     1 /* Get fildes flags */
#define F_SETFD     2 /* Set fildes flags */
#define F_GETFL     3 /* Get file flags */
#define F_SETFL     4 /* Set file flags */
#define F_GETLK     5 /* Get file lock */
#define F_SETLK     6 /* Set file lock */
#define F_SETLKW    7 /* Set file lock and wait */
#define F_CHKFL     8 /* Check legality of file flag changes */

/*file segment locking set data type - information passed to system by user*/
struct flock {
    short    l_type;
    short    l_whence;
    long     l_start;
    long     l_len; /* len = 0 means until end of file */
    short    l_sysid;
    short    l_pid;
};

/* file segment locking types */
/* Read lock */
#define F_RDLCK    01
/* Write lock */
#define F_WRLCK    02
/* Remove lock(s) */
#define F_UNLCK    03

```

Figure 3-1: The **fcntl.h** Header File

The format of the **fcntl(2)** system call is

```
int fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

fildes is the file descriptor returned by the **open** system call. In addition to defining tags that are used as the commands on **fcntl** system calls, **fcntl.h** includes the declaration for a *struct flock* that is used to pass values that control where locks are to be placed.

lockf

A subroutine, **lockf(3)**, can also be used to lock sections of a file or an entire file. The format of **lockf** is:

```
#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

fildes is the file descriptor; *function* is one of four control values defined in **unistd.h** that let you lock, unlock, test and lock, or simply test to see if a lock is already in place. *size* is the number of contiguous bytes to be locked or unlocked. The section of contiguous bytes can be either forward or backward from the current offset in the file. (You can arrange to be somewhere in the middle of the file by using the **lseek(2)** system call.)

Where to Find More Information

There is an example of file and record locking in the sample application at the end of this chapter. The manual pages that apply to this facility are **fcntl(2)**, **fcntl(5)**, **lockf(3)**, and **chmod(2)** in the *System V Reference Manual*. Chapter 7 is a detailed discussion of the subject with a number of examples.

Interprocess Communications

In Chapter 2 we described **forking** and **execing** as methods of communicating between processes. Business applications running on a UNIX system computer often need more sophisticated methods. In applications, for example, where fast response is critical, a number of processes may be brought up at the start of a business day to be constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. To go back to the ticket reservation example again for a moment, if a customer calls to reserve a seat on the 5 o'clock flight to Detroit, you don't want to have to say, "Yes, sir. Just hang on a minute while I start up the reservations program." In transaction driven systems, the normal mode of processing is to have all the components of the application standing by waiting for some sort of an indication that there is work to do.

To meet requirements of this type the UNIX system offers a set of nine system calls and their accompanying header files, all under the umbrella name of Interprocess Communications (IPC).

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes:

- | | |
|------------|---|
| messages | communication is in the form of data stored in a buffer. The buffer can be either sent or received. |
| semaphores | communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25. |

shared memory communication takes place through a common area of main memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there.

The sets of IPC system calls are:

msgget	semget	shmget
msgctl	semctl	shmctl
msgop	semop	shmop

IPC get Calls

The **get** calls each return to the calling program an identifier for the type of IPC facility that is being requested.

IPC ctl Calls

The **ctl** calls provide a variety of control operations that include obtaining (IPC_STAT), setting (IPC_SET) and removing (IPC_RMID), the values in data structures associated with the identifiers picked up by the **get** calls.

IPC op Calls

The **op** manual pages describe calls that are used to perform the particular operations characteristic of the type of IPC facility being used. **msgop** has calls that send or receive messages. **semop** (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. **shmop** has calls that attach or detach shared memory segments.

Where to Find More Information

An example of the use of some IPC features is included in the sample application at the end of this chapter. The system calls are all located in Section (2) of the *System V Reference Manual*. Don't overlook **intro(2)**. It includes descriptions of the data structures that are used by IPC facilities. A detailed description of IPC, with many code examples that use the IPC system calls, is contained in Chapter 8.

Programming Terminal Screens

The facility for setting up terminal screens to meet the needs of your application is provided by two parts of the UNIX system. The first of these, **terminfo**, is a data base of compiled entries that describe the capabilities of terminals and the way they perform various operations.

The **terminfo** data base normally begins at the directory **/usr/lib/terminfo**. The members of this directory are themselves directories, generally with single-character names that are the first character in the name of the terminal. The compiled files of operating characteristics are at the next level down the hierarchy. For example, the standard entry for a terminal on a SUPERMAX is located in **usr/lib/terminfo/T/T3-24-C80**.

The Virtual Terminal Interface (VTI) at the SUPERMAX use description files placed in **/etc/types**. The Terminal Interface will work correct if the **terminology**-program is invoked with the description file as an argument.

Describing the capabilities of a terminal can be a painstaking task. Quite a good selection of terminal entries is included in the **/etc/types** that comes with your SUPERMAX Computer. However, if you have a type of terminal that is not already described in the data base, the best way to proceed is to find a description of one that comes close to having the same capabilities as yours and building on that one.

For further information about the SUPERMAX VTI please refer to the Supermax Virtual Interface Guide.

curses

After you have made sure that the terminology has been executed with correct argument, you can then proceed to use the routines that make up the **curses(3X)** package to create and manage screens for your application.

The **curses** library includes functions to:

- define portions of your terminal screen as windows
- define pads that extend beyond the borders of your physical terminal screen and let you see portions of the pad on your terminal
- read input from a terminal screen into a program
- write output from a program to your terminal screen
- manipulate the information in a window in a virtual screen area and then send it to your physical screen

Where to Find More Information

In the sample application at the end of this chapter, we show how you might use **curses** routines. Chapter 9 contains a tutorial on the subject. The manual pages for **curses** are in Section (3X), and those for **terminfo** are in Section (4) and those for **terminology** in Section (1) of the *System V Reference Manual*.



This page is intentionally left blank

Programming Support Tools

This section covers UNIX system components that are part of the programming environment, but that have a highly specialized use. We refer to such things as:

- link edit command language
- Common Object File Format
- libraries
- Symbolic Debugger
- **lint** as a portability tool

Link Edit Command Language

The link editor command language is for use when the default arrangement of the **ld** output will not do the job. The default locations for the standard Common Object File Format sections are described in **a.out(4)** in the *System V Reference Manual*. On a SUPERMAX Computer, it depends on the processor type where the different parts of data are loaded. Also the stack position is processor dependant, but it will always grow to lower memory addresses.

The link editor command language provides directives for describing different arrangements. The two major types of link editor directives are MEMORY and SECTIONS. MEMORY directives can be used to define the boundaries of configured and unconfigured sections of memory within a machine, to name sections, and to assign specific attributes (read, write, execute, and initialize) to portions of memory. SECTIONS directives, among a lot of other functions, can be used to bind sections of the object file to specific addresses within the configured portions of memory.

Why would you want to be able to do those things? Well, the truth is that in the majority of cases you don't have to worry about it. The need to control the link editor output becomes more urgent under two, possibly related, sets of circumstances.

1. Your application is large and consists of a lot of object files.
2. The hardware your application is to run on is tight for space.

Where to Find More Information

Chapter 11 gives a detailed description of the subject.

Common Object File Format

The details of the Common Object File Format have never been looked on as stimulating reading. In fact, they have been recommended to hard-core insomniacs as preferred bedtime fare. However, if you're going to break into the ranks of really sophisticated UNIX system programmers, you're going to have to get a good grasp of COFF. A knowledge of COFF is fundamental to using the link editor command language. It is also good background knowledge for tasks such as:

- setting up archive libraries or shared libraries
- using the Symbolic Debugger

The following system header files contain definitions of data structures of parts of the Common Object File Format:

<syms.h>	symbol table format
<linenum.h>	line number entries
<ldfcn.h>	COFF access routines
<filehdr.h>	file header for a common object file
<a.out.h>	common assembler and link editor output
<scnhdr.h>	section header for a common object file
<reloc.h>	relocation information for a common object file
<storclass.h>	storage classes for common object files

The object file access routines are described below under the heading "The Object File Library."

Where to Find More Information

Chapter 10 gives a detailed description of COFF.

Libraries

A library is a collection of related object files and/or declarations that simplify programming effort. Programming groups involved in the development of applications often find it convenient to establish private libraries. For example, an application with a number of programs using a common data base can keep the I/O routines in a library that is searched at link edit time.

Prior to Release 3.1 of the UNIX System V the libraries, whether system supplied or application developed, were collections of common object format files stored in an archive (*filename.a*) file that was searched by the link editor to resolve references. Files in the archive that were needed to satisfy unresolved references became a part of the resulting executable.

In Chapter 2 we described many of the functions that are found in the standard C library, **libc.a**. The next two sections describe two other libraries, the object file library and the math library.

The Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. The need to work at this level of detail with object files occurs most often in the development of new tools that manipulate object files. For a description of the format of an object file, see "The Common Object File Format" in Chapter 10. This library consists of several portions. The functions reside in:

```
/usr/lib68020/libld.a,  
/usr/libR3KMO/libmld.a,  
or in  
/usr/libR3KMI/libmld.a
```

and are loaded during the compilation of a C language program by the `-l` command line option:

```
cc file -lld
```

which causes the link editor to search the object file library. The argument `-lld` must appear after all files that reference functions in `libld.a`.

The following header files must be included in the source code.

```
#include <stdio.h>  
#include <a.out.h>  
#include <ldfcn.h>
```

Function	Reference	Brief Description
ldaclose	ldclose(3X)	Close object file being processed.
ldahread	ldahread(3X)	Read archive header.
ldaopen	ldopen(3X)	Open object file for reading.
ldclose	ldclose(3X)	Close object file being processed.
ldfhread	ldfhread(3X)	Read file header of object file being processed.
ldgetname	ldgetname(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	ldlread(3X)	Prepare object file for reading line number entries via ldlitem .
ldlitem	ldlread(3X)	Read line number entry from object file after ldlinit .
ldlread	ldlread(3X)	Read line number entry from object file.
ldlseek	ldlseek(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	ldlseek(3X)	Seeks to the line number entries of the object file being processed given the name of a section.
ldnrseek	ldrseek(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	ldshread(3X)	Read section header of the named section of the object file being processed.
ldnsseek	ldsseek(3X)	Seeks to the section of the object file being processed given the name of a section.

Function	Reference	Brief Description
ldohseek	ldohseek(3X)	Seeks to the optional file header of the object file being processed.
ldopen	ldopen(3X)	Open object file for reading.
ldrseek	ldrseek(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	ldshread(3X)	Read section header of an object file being processed.
ldsseek	ldsseek(3X)	Seeks to the section of the object file being processed.
ldtbindex	ldtbindex(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	ldtbread(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	ldtbseek(3X)	Seeks to the symbol table of the object file being processed.
sgetl	sputl(3X)	Access long integer data in a machine independent format.
sputl	sputl(3X)	Translate a long integer into a machine independent format.

156

Common Object File Interface Macros (**ldfcn.h**)

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, which is in the header file **ldfcn.h** (see **ldfcn(4)**). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen(3X)** allocates and initializes the LDFILE structure and returns a pointer to the structure. The fields of the LDFILE structure may be accessed individually through the following macros:

- The **TYPE** macro returns the magic number of the file, which is used to distinguish between archive files and object files that are not part of an archive.
- The **IOPTR** macro returns the file pointer, which was opened by **ldopen(3X)** and is used by the input/output functions of the C library.
- The **OFFSET** macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file.
- The **HEADER** macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an LDFILE structure into a reference to its file descriptor field. The available macros are described in **ldfcn(4)** in the *System V Reference Manual*.

The Math Library

The math library package consists of functions and a header file. The functions are located and loaded during the compilation of a C language program by the **-l** option on a command line, as follows:

```
cc file -lm
```

This option causes the link editor to search the math library, **libm.a**. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

Programming Support Tools

near the beginning of each file that uses the routines.

The functions are grouped into the following categories:

- trigonometric functions
- Bessel functions
- hyperbolic functions
- miscellaneous functions

Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double-precision.

Function	Reference	Brief Description
acos	trig(3M)	Return arc cosine.
asin	trig(3M)	Return arc sine.
atan	trig(3M)	Return arc tangent.
atan2	trig(3M)	Return arc tangent of a ratio.
cos	trig(3M)	Return cosine.
sin	trig(3M)	Return sine.
tan	trig(3M)	Return tangent.

Bessel Functions

These functions calculate Bessel functions of the first and second kinds of several orders for real values. The Bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel(3M)**.

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

Function	Reference	Brief Description
cosh	sinh(3M)	Return hyperbolic cosine.
sinh	sinh(3M)	Return hyperbolic sine.
tanh	sinh(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers.

Function	Reference	Brief Description
ceil	floor(3M)	Returns the smallest integer not less than a given value.
exp	exp(3M)	Returns the exponential function of a given value.
fabs	floor(3M)	Returns the absolute value of a given value.
floor	floor(3M)	Returns the largest integer not greater than a given value.
fmod	floor(3M)	Returns the remainder produced by the division of two given values.
gamma	gamma(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.

Function	Reference	Brief Description
hypot	hypot(3M)	Return the square root of the sum of the squares of two numbers.
log	exp(3M)	Returns the natural logarithm of a given value.
log10	exp(3M)	Returns the logarithm base ten of a given value.
matherr	matherr(3M)	Error-handling function.
pow	exp(3M)	Returns the result of a given value raised to another given value.
sqrt	exp(3M)	Returns the square root of a given value.

Symbolic Debugger

The use of **sdb/dbx** was mentioned briefly in Chapter 2. In this section we want to say a few words about **sdb/dbx** within the context of an application development project.

sdb/dbx works on a process, and enables a programmer to find errors in the code. It is a tool a programmer might use while coding and unit testing a program, to make sure it runs according to its design. **sdb/dbx** would normally be used prior to the time the program is turned over, along with the rest of the application, to testers. During this phase of the application development cycle programs are compiled with the **-g** option of **cc** to facilitate the use of the debugger. The symbol table should not be stripped from the object file. Once the programmer is satisfied that the program is error-free, **strip(1)** can be used to reduce the file storage overhead taken by the file.

Where to Find More Information

Chapter 14 contains information on how to use **sdb/dbx**. The manual page is in Section (1) of the *System V Reference Manual*.

lint as a Portability Tool

It is a characteristic of the UNIX system that language compilation systems are somewhat permissive. Generally speaking it is a design objective that a compiler should run fast. Most C compilers, therefore, let some things go unflagged as long as the language syntax is observed statement by statement. This sometimes means that while your program may run, the output will have some surprises. It also sometimes means that while the program may run on the machine on which the compilation system runs, there may be real difficulties in running it on some other machine.

That's where **lint** comes in. **lint** produces comments about inconsistencies in the code. The types of anomalies flagged by **lint** are:

- cases of disagreement between the type of value expected from a called function and what the function actually returns
- disagreement between the types and number of arguments expected by functions and what the function receives
- inconsistencies that might prove to be bugs
- things that might cause portability problems

Here is an example of a portability problem that would be caught by **lint**.

Code such as this:

```
int i = lseek(fdes, offset, whence)
```

would get by most compilers. However, **lseek** returns a long integer representing the address of a location in the file. On a machine with a 16-bit integer and a bigger **long int**, it would produce incorrect

Programming Support Tools

results, because `i` would contain only the last 16 bits of the value returned.

Since it is reasonable to expect that an application written for a UNIX system machine will be able to run on a variety of computers, it is important that the use of `lint` be a regular part of the application development.

Where to Find More Information

Chapter 15 contains a description of `lint` with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *System V Reference Manual*.

Project Control Tools

Volumes have been written on the subject of project control. It is an item of top priority for the managers of any application development team. Two UNIX system tools that can play a role in this area are described in this section.

make

make is extremely useful in an application development project for keeping track of what object files need to be recompiled as changes are made to source code files. One of the characteristics of programs in a UNIX system environment is that they are made up of many small pieces, each in its own object file, that are link edited together to form the executable file. Quite a few of the UNIX system tools are devoted to supporting that style of program architecture. For example, archive libraries, shared libraries and even the fact that the **cc** command accepts **.o** files as well as **.c** files, and that it can stop short of the **ld** step and produce **.o** files instead of an **a.out**, are all important elements of modular architecture. The two main advantages of this type of programming are that

- A file that performs one function can be re-used in any program that needs it.
- When one function is changed, the whole program does not have to be recompiled.

On the flip side, however, a consequence of the proliferation of object files is an increased difficulty in keeping track of what does need to be recompiled, and what doesn't. **make** is designed to help deal with this problem. You use **make** by describing in a specification file, called **makefile**, the relationship (that is, the dependencies) between the different files of your program. Once having done that, you conclude a session in which possibly a number of your source code files have been changed by running the **make** command. **make** takes

Project Control Tools

care of generating a new **a.out** by comparing the time-last-changed of your source code files with the dependency rules you have given it.

make has the ability to work with files in archive libraries or under control of the Source Code Control System (SCCS).

Where to Find More Information

The **make**(1) manual page is contained in the *System V Reference Manual*. Chapter 12 gives a complete description of how to use **make**.

SCCS

SCCS is an acronym for Source Code Control System. It consists of a set of 14 commands used to track evolving versions of files. Its use is not limited to source code; any text files can be handled, so an application's documentation can also be put under control of SCCS. SCCS can:

- store and retrieve files under its control
- allow no more than a single copy of a file to be edited at one time
- provide an audit trail of changes to files
- reconstruct any earlier version of a file that may be wanted

SCCS files are stored in a special coded format. Only through commands that are part of the SCCS package can files be made available in a user's directory for editing, compiling, etc. From the point at which a file is first placed under SCCS control, only changes to the original version are stored. For example, let's say that the program, **restate**, that was used in several examples in Chapter 2, was controlled by SCCS. One of the original pieces of that program is a file called **oppty.c** that looks like this:

```

                                /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
    return(ps->i/12 * ps->t * ps->dp);
}

```

If you decide to add a message to this function, you might change the file like this:

```

                                /* Opportunity Cost -- oppty.c */
#include "recdef.h"
#include <stdio.h>

float
oppty(ps)
struct rec *ps;
{
    (void) fprintf(stderr, "Opportunity calling\n");
    return(ps->i/12 * ps->t * ps->dp);
}

```

SCCS saves only the two new lines from the second version, with a coded notation that shows where in the text the two lines belong. It also includes a note of the version number, lines deleted, lines inserted, total lines in the file, the date and time of the change and the login id of the person making the change.

Project Control Tools

Where to Find More Information

Chapter 13 is an SCCS user's guide. SCCS commands are in Section (1) of the *System V Reference Manual*.

liber, A Library System

To illustrate the use of UNIX system programming tools in the development of an application, we are going to pretend we are engaged in the development of a computer system for a library. The system is known as **liber**. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficacy of this design. It is the way it is only in order to provide some passably realistic examples of UNIX system programming tools in use.

liber is a system for keeping track of the books in a library. The hardware consists of a single computer with terminals throughout the library. One terminal is used for adding new books to the data base. Others are used for checking out books and as electronic card catalogs.

The design of the system calls for it to be brought up at the beginning of the day and remain running while the library is in operation. The system has one master index that contains the unique identifier of each title in the library. When the system is running the index resides in memory. Semaphores are used to control access to the index. In the pages that follow fragments of some of the system's programs are shown to illustrate the way they work together. The startup program performs the system initialization; opening the semaphores and shared memory; reading the index into the shared memory; and kicking off the other programs. The id numbers for the shared memory and semaphores (**shmid**, **wrtsem**, and **rdsem**) are read from a file during initialization. The programs all share the in-memory index. They attach it with the following code:

SAMPLE APPLICATION: liber

```

/* attach shared memory for index */
if ((int)(index = (INDEX *) shmat(shmid, NULL, 0)) == -1)
{
    (void) fprintf(stderr, "shmat failed: %d\n", errno);
    exit(1);
}

```

Of the programs shown, **add-books** is the only one that alters the index. The semaphores are used to ensure that no other programs will try to read the index while **add-books** is altering it. The checkout program locks the file record for the book, so that each copy being checked out is recorded separately and the book cannot be checked out at two different checkout stations at the same time.

The program fragments do not provide any details on the structure of the index or the book records in the data base.

```

/* liber.h - header file for the library system. */

typedef ... INDEX;      /* data structure for book file index */
typedef struct {       /* type of records in book file */
    char title[30];
    char author[30];
    .
    .
    .
} BOOK;
int shmid;
int wrtsem;
int rdsem;
INDEX *index;

int book_file;
BOOK book_buf;

```

(continued on next page)


```
/* startup program*/

/*
 * 1. Open shared memory for file index and read it in.
 * 2. Open two semaphores for providing exclusive write access to index.
 * 3. Stash id's for shared memory segment and semaphores in a file
 *    where they can be accessed by the programs.
 * 4. Start programs: add-books, card-catalog, and checkout running
 *    on the various terminals throughout the library.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;

key_t key;
int shmid;
int wrtsem;
int rdsem;
FILE *ipc_file;

main()
{
    .
    .
    .
    if ((shmid = shmget(key, sizeof(INDEX), IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: shmget failed: errno=%d\n",
            errno);
        exit(1);
    }
    if ((wrtsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: semget failed: errno=%d\n",
```

(continued on next page)

SAMPLE APPLICATION: liber

```
    errno);
    exit(1);
}
if ((rdsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
{
    (void) fprintf(stderr, "startup: semget failed: errno=%d\n",
        errno);
    exit(1);
}
(void) fprintf(ipc_file, "%d\n%d\n%d\n", shmids, wrtsem, rdsem);

/*
 * Start the add-books program running on the terminal in the
 * basement. Start the checkout and card-catalog programs
 * running on the various other terminals throughout the library.
 */
.
.
.
}

/* card-catalog program */

/*
 * 1. Read screen for author and title.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. Print book record on screen or indicate book was not found.
 * 5. Go to 1.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
```

(continued on next page)

```
main() {
    .
    .
    while (1)
    {
        /*
         * Read author/title/subject information from screen.
         */
        /*
         * Wait for write semaphore to reach 0 (index not being
         written).
         */

        sop[0].sem_op = 1;
        if (semop(wrtsem, sop, 1) == -1)
        {
            (void) fprintf(stderr, "semop failed: %d\n",
                errno);
            exit(1);
        }
        /*
         * Increment read semaphore so potential writer will wait
         * for us to finish reading the index.
         */
        sop[0].sem_op = 0;
        if (semop(rdsem, sop, 1) == -1)
        {
            (void) fprintf(stderr, "semop failed: %d\n",
                errno);
            exit(1);
        }

        /* Use index to find file pointer(s) for book(s) */

        /* Decrement read semaphore */
        sop[0].sem_op = -1;
        if (semop(rdsem, sop, 1) == -1)
        {
            (void) fprintf(stderr, "semop failed: %d\n",
                errno);
            exit(1);
        }
    }
}
```

(continued on next page)

SAMPLE APPLICATION: liber

```
/*
 * Now we use the file pointers found in the index to
 * read the book file. Then we print the information
 * on the book(s) to the screen.
 */
} /* while */
}

/* checkout program */

/*
 * 1. Read screen for Dewey Decimal number of book to be checked out.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. If book not found print message on screen, otherwise lock
 *    book record and read.
 * 5. If book already checked out print message on screen, otherwise
 *    mark record "checked out" and write back to book file.
 * 6. Unlock book record.
 * 7. Go to 1.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
long lseek();
extern int errno;
struct flock flk;
struct sembuf sop[1];
long bookpos;

main()
{
    .
    .
    .
}
```

(continued on next page)

```

while (1)
{
    /*
    * Read Dewey Decimal number from screen.
    */
    /*
    * Wait for write semaphore to reach 0 (index not being
    written).
    */
    sop[0].sem_flg = 0;
    sop[0].sem_op = 0;
    if (semop(wrtsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n",
            errno);
        exit(1);
    }
    /*
    * Increment read semaphore so potential writer will wait
    * for us to finish reading the index.
    */
    sop[0].sem_op = 1;
    if (semop(rdsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n",
            errno);
        exit(1);
    }
    /*
    * Now we can use the index to find the book's record position.
    * Assign this value to "bookpos".
    */

    /* Decrement read semaphore */
    sop[0].sem_op = -1;
    if (semop(rdsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n",
            errno);
        exit(1);
    }
}
    
```

(continued on next page)

SAMPLE APPLICATION: **liber**

```
    }

    /* Lock the book's record in book file, read the record. */
    flk.l_type = F_WRLCK;
    flk.l_whence = 0;
    flk.l_start = bookpos;
    flk.l_len = sizeof(BOOK);
    if (fcntl(book_file, F_SETLKW, &flk) == -1)
    {
        (void) fprintf(stderr, "trouble locking: %d\n",
            errno);
        exit(1);
    }
    if (lseek(book_file, bookpos, 0) == -1)
    {
        Error processing for lseek;
    }
    if (read(book_file, &book_buf, sizeof(BOOK)) == -1)
    {
        Error processing for read;
    }

    /*
     * If the book is checked out inform the client, otherwise
     * mark the book's record as checked out and write it
     * back into the book file.
     */

    /* Unlock the book's record in book file. */
    flk.l_type = F_UNLCK;
    if (fcntl(book_file, F_SETLK, &flk) == -1)
    {
        (void) fprintf(stderr, "trouble unlocking: %d\n",
            errno);
        exit(1);
    }
} /* while */
}
/* add-books program */
/*
 * 1. Read a new book entry from screen.
 * 2. Insert book in book file.
```

(continued on next page)

```
* 3. Use semaphore "wrtsem" to block new readers.
* 4. Wait for semaphore "rdsem" to reach 0.
* 5. Insert book into index.
* 6. Decrement wrtsem.
* 7. Go to 1.
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
BOOK bookbuf;

main()
{
    .
    .
    for (;;)
    {
        /*
         * Read information on new book from screen.
         */
        addscr(&bookbuf);
        /* write new record at the end of the bookfile.
         * Code not shown, but
         * addscr() returns a 1 if title information has
         * been entered, 0 if not.
         */
        /*
         * Increment write semaphore, blocking new readers from
         * accessing the index.
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 1;
        if (semop(wrtsem, sop, 1) == -1)
        {
            (void) fprintf(stderr, "semop failed: %d\n",
                errno);
        }
    }
}
```

(continued on next page)

SAMPLE APPLICATION: `liber`

```
        exit(1);
    }
    /*
     * Wait for read semaphore to reach 0 (all readers to finish
     * using the index).
     */
    sop[0].sem_op = 0;
    if (semop(rdsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n",
            errno);
        exit(1);
    }
    /*
     * Now that we have exclusive access to the index we
     * insert our new book with its file pointer.
     */

    /* Decrement write semaphore, permitting readers to read index. */
    sop[0].sem_op = -1;
    if (semop(wrtsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n",
            errno);
        exit(1);
    }
    } /* for */
}
```

The example following, `addscr()`, illustrates two significant points about **curses** screens:

1. Information read in from a **curses** window can be stored in fields that are part of a structure defined in the header file for the application.

2. The address of the structure can be passed from another function where the record is processed.

```

/* addscr is called from add-books.
 * The user is prompted for title
 * information.
 */

#include <curses.h>
WINDOW *cmdwin;
addscr(bb)
struct BOOK *bb;
{

    int c;
    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(6, 40, 3, 20);
    mvprintw(0, 0, "This screen is for adding titles to the data base");
    mvprintw(1, 0, "Enter a to add; q to quit: ");
    refresh();
    for (;;)
    {
        refresh();
        c = getch();
        switch (c) {
            case 'a':
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter title: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->title);
                noecho();
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter author: ");
                wmove(cmdwin, 2, 1);
                echo();

```

(continued on next page)

SAMPLE APPLICATION: liber

```
wrefresh(cmdwin);
wgetstr(cmdwin, bb->author);
noecho();
werase(cmdwin);
wrefresh(cmdwin);
endwin();
return(1);

case 'q':
    erase();
    endwin();
    return(0);
}
}
}

#
# Makefile for liber library system
#

CC = cc
CFLAGS = -O
all: startup add-books checkout card-catalog

startup: liber.h startup.c
    $(CC) $(CFLAGS) -o startup startup.c

add-books: add-books.o addscr.o
    $(CC) $(CFLAGS) -o add-books add-books.o addscr.o

add-books.o: liber.h

checkout: liber.h checkout.c
    $(CC) $(CFLAGS) -o checkout checkout.c

card-catalog: liber.h card-catalog.c
    $(CC) $(CFLAGS) -o card-catalog card-catalog.c
```

Chapter 4: awk

	Page
Introduction.....	4- 1
Basic awk	4- 3
Program Structure	4- 3
Usage	4- 4
Fields.....	4- 5
Printing	4- 6
Formatted Printing	4- 8
Simple Patterns	4- 9
Simple Actions	4-10
Built-in Variables	4-10
User-defined Variables.....	4-11
Functions	4-11
A Handful of Useful One-liners	4-12
Error Messages	4-13
Patterns	4-15
BEGIN and END	4-15
Relational Expressions.....	4-16
Regular Expressions.....	4-17
Combinations of Patterns.....	4-21
Pattern Ranges	4-22
Actions	4-23
Built-in Variables.....	4-23
Arithmetic	4-24
Strings and String Functions.....	4-27

Table of Contents

	Page
Field Variables	4 - 31
Number or String?	4 - 32
Control Flow Statements	4 - 34
Arrays	4 - 37
User-Defined Functions	4 - 40
Some Lexical Conventions	4 - 41
Output	4 - 43
The print Statement	4 - 43
Output Separators	4 - 43
The printf Statement	4 - 44
Output into Files	4 - 46
Output into Pipes	4 - 46
Input	4 - 49
Files and Pipes	4 - 49
Input Separators	4 - 50
Multi-line Records	4 - 50
The getline Function	4 - 51
Command-line Arguments	4 - 54
Using awk with Other Commands and the Shell	4 - 57
The system Function	4 - 57
Cooperation with the Shell	4 - 57
Example Applications	4 - 61
Generating Reports	4 - 61
Additional Examples	4 - 63
Word Frequencies	4 - 63
Accumulation	4 - 64
Random Choice	4 - 64

Table of Contents

	Page
Shell Facility.....	4-65
Form-letter Generation.....	4-65
awk Summary.....	4-67
Command Line.....	4-67
Patterns.....	4-67
Control Flow Statements.....	4-67
Input-output.....	4-68
Functions.....	4-68
String Functions.....	4-69
Arithmetic Functions.....	4-69
Operators (Increasing Precedence).....	4-70
Regular Expressions (Increasing Precedence).....	4-70
Built-in Variables.....	4-71
Limits.....	4-71
Initialization, Comparison, and Type Coercion.....	4-72

Table of Contents

This page is intentionally left blank

Introduction

NOTE

This chapter describes the new version of **awk** released in UNIX System V Release 3.1 and described in **awk(1)**. An earlier version is described in **oawk(1)**.

Suppose you want to tabulate some survey results stored in a file, print various reports summarizing these results, generate form letters, reformat a data file for one application package to use with another package, or count the occurrences of a string in a file. **awk** is a programming language that makes it easy to handle these and many other tasks of information retrieval and data processing. The name **awk** is an acronym constructed from the initials of its developers; it denotes the language and also the UNIX system command you use to run an **awk** program.

awk is an easy language to learn. It automatically does quite a few things that you have to program for yourself in other languages. As a result, many useful **awk** programs are only one or two lines long. Because **awk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **awk** is also a good language for prototyping.

The first part of this chapter introduces you to the basics of **awk** and is intended to make it easy for you to start writing and running your own **awk** programs. The rest of the chapter describes the complete language and is somewhat less tutorial. For the experienced **awk** user, there's a summary of the language at the end of the chapter.

You should be familiar with the UNIX system and shell programming to use this chapter. Although you don't need other programming experience, some knowledge of the C programming language is beneficial, because many constructs found in **awk** are also found in C.

This page is intentionally left blank

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

Program Structure

The basic operation of **awk**(1) is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **awk** program is a sequence of pattern-action statements, as Figure 4-1 shows.

Structure:

```
pattern    { action }  
pattern    { action }  
...
```

Example:

```
$1 == "address" { print $2, $3 }
```

Figure 4-1: **awk** Program Structure and Example

The example in the figure is a typical **awk** program, consisting of one pattern-action statement. The program prints the second and third fields of each input line whose first field is address. In general, **awk** programs work by matching each line of input against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read.

Basic awk

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in

```
$1 == "name"
```

the matching line is printed. If there is no pattern with an action, as in

```
{ print $1, $2 }
```

the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Usage

There are two ways to run an **awk** program. First, you can type the command line

```
awk 'pattern-action statements' optional list of input files
```

to execute the pattern-action statements on the set of named input files. For example, you could say

```
awk '{ print $1, $2 }' file1 file2
```

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like **\$** from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk(1)** reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (-) as one of the input files. For example,

```
awk '{ print $3, $4 }' file1 -
```

says to read input first from **file1** and then from the standard input.

The arrangement above is convenient when the **awk** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the **-f** option to fetch it:

```
awk -f program file optional list of input files
```

For example, the following command line says to fetch and execute **myprogram** on input from the file **file1**:

```
awk -f myprogram file1
```

Fields

awk normally reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline. **awk** then splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the file **countries**, which contains information about the ten largest countries in the world. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent on which it is found. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The white space between fields is a tab in the original input; a single blank separates North and South from America .

Basic awk

USSR	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

Figure 4-2: The Sample Input File **countries**

This file is typical of the kind of data **awk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, so that the first record of **countries** would have four fields, the second five, and so on. It's possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs. The first field within a line is called **\$1**, the second **\$2**, and so forth. The entire record is called **\$0**.

Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **awk** program consisting of a single **print** statement

```
{ print }
```

so the command line

```
awk '{ print }' countries
```

prints each line of **countries**, copying the file to the standard output. The **print** statement can also be used to print parts of a record; for instance, the program

```
{ print $1, $3 }
```

prints the first and third fields of each record. Thus

```
awk '{ print $1, $3 }' countries
```

produces as output the sequence of lines:

```
USSR 262  
Canada 24  
China 866  
USA 219  
Brazil 116  
Australia 14  
India 637  
Argentina 26  
Sudan 19  
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator, which by default is a single blank. Each line printed is terminated by the output record separator, which by default is a newline.

NOTE

In the remainder of this chapter, we only show **awk** programs, without the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the **awk** command, or by putting it in a file and invoking **awk** with the **-f** flag, as discussed in "awk Command Usage." In an example, if no input is mentioned, the input is assumed to be the file **countries**.

Basic awk

Formatted Printing

For more carefully formatted output, **awk** provides a C-like **printf** statement

```
printf format, expr1, expr2, . . . , exprn
```

which prints the *expr_i*'s according to the specification in the string *format*. For example, the **awk** program

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field (*\$1*) as a string of 10 characters (right justified), then a space, then the third field (*\$3*) as a decimal number in a six-character field, then a newline (*\n*). With input from the file **countries**, this program prints an aligned table:

USSR	262
Canada	24
China	866
USA	219
Brazil	116
Australia	14
India	637
Argentina	26
Sudan	19
Algeria	18

With **printf**, no output separators or newlines are produced automatically; you must create them yourself by using *\n* in the format specification. "The **printf** Statement" in this chapter contains a full description of **printf**.

Simple Patterns

You can select specific records for printing or other processing by using simple patterns. **awk** has three kinds of patterns. First, you can use patterns called relational expressions that make comparisons. For example, the operator tests for equality. To print the lines for which the fourth field equals the string *Asia*, we can use the program consisting of the single pattern

```
$4 == "Asia"
```

With the file **countries** as input, this program yields

```
USSR  8650  262  Asia
China 3692  866  Asia
India 1269  637  Asia
```

The complete set of comparisons is $>$, $>=$, $<$, $<=$, $=$ (equal to) and $!=$ (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with a population greater than 100 million. The program

```
$3 > 100
```

is all that is needed. (Remember that the third field in the file **countries** is the population in millions.) It prints all lines in which the third field exceeds 100.

Second, you can use patterns called regular expressions that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters *US* anywhere; with the file **countries** as input, it prints

```
USSR  8650  262  Asia
USA    3615  219  North America
```

We will have a lot more to say about regular expressions later in this chapter.

Basic awk

Third, you can use two special patterns, **BEGIN** and **END**, that match before the first record has been read and after the last record has been processed. This program uses **BEGIN** to print a title:

```
BEGIN { print "Countries of Asia:" }
/Asia/ { print "    ", $1 }
```

The output is

```
Countries of Asia:
    USSR
    China
    India
```

Simple Actions

We have already seen the simplest action of an **awk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **awk**(1) counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields in the record. So the program

```
{ print NR, NF }
```

prints the number of each line and how many fields it has, while

```
{ print NR, $0 }
```

prints each record preceded by its record number.

User-defined Variables

Besides providing built-in variables like **NF** and **NR**, **awk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file **countries**:

```
    { sum = sum + $3 }
END  { print "Total population is", sum, "million"
      print "Average population of", NR,
          "countries is", sum/NR
      }
```

NOTE

awk initializes **sum** to zero before it is used.

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

awk has built-in functions that handle common arithmetic and string operations for you. For example, there's an arithmetic function that computes square roots. There is also a string function that substitutes one string for another. **awk** also lets you define your own functions. Functions are described in detail in the section "Actions" in this chapter.

A Handful of Useful One-liners

Although **awk** can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. Here is a collection of other short programs that you may find useful and instructive. They are not explained here, but any new constructs do appear later in this chapter.

Print last field of each input line:

```
{ print $NF }
```

Print 10th input line:

```
NR == 10
```

Print last input line:

```
{ line = $0 }
END { print line }
```

Print input lines that don't have four fields:

```
NF != 4 { print $0, "does not have 4 fields" }
```

Print input lines with more than four fields:

```
NF > 4
```

Print input lines with last field more than 4:

```
$NF > 4
```

Print total number of input lines:

```
END { print NR }
```

Print total number of fields:

```
{ nf = nf + NF }
END { print nf }
```

Print total number of input characters:

```
{ nc = nc + length($0) }
END { print nc + NR }
```

(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
/Asia/ { nlines++ }  
END    { print nlines }
```

(The statement `nlines++` has the same effect as `nlines = nlines + 1`.)

Error Messages

If you make an error in your **awk** program, you generally get an error message. For example, trying to run the program

```
$3 < 200 { print ( $1 ) }
```

generates the error messages

```
awk: syntax error at source line 1  
context is  
    $3 < 200 { print ( >>> $1 ) <<<  
awk: illegal statement at source line 1  
1 extra (
```

Some errors may be detected while your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (**NR**) and the line number in the program.

This page is intentionally left blank

PATTERNS

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

BEGIN and **END** are two special patterns that give you a way to control initialization and wrap-up in an **awk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done once, before the **awk** command starts to read its first input record. The pattern **END** matches the end of the input, after the last record has been processed.

The following **awk** program uses **BEGIN** to set the field separator to tab (`\t`) and to put column headings on the output. The field separator is stored in a built-in variable called **FS**. Although **FS** can be reset at any time, usually the only sensible place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. (Notice that a long line can be continued after a comma.)

```
BEGIN {FS = "\t"
        printf "%10s %6s %5s   %s\n",
               "COUNTRY", "AREA", "POP", "CONTINENT" }
{printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
  area = area + $2; pop = pop + $3 }
END   {printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file **countries** as input, this program produces:

PATTERNS

COUNTRY	AREA	POP	CONTINENT
USSR	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa
TOTAL	30292	2201	

Relational Expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. **awk** has six relational operators, and two regular expression matching operators, `~` (tilde) and `!~`, which are discussed in the next section, for making comparisons. Figure 4-3 shows these operators and their meanings.

Operator	Meaning
<code><</code>	less than
<code>< =</code>	less than or equal to
<code>= =</code>	equal to
<code>!=</code>	not equal to
<code>> =</code>	greater than or equal to
<code>></code>	greater than
<code>~</code>	matches
<code>!~</code>	does not match

Figure 4-3: **awk** Comparison Operators

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can tell what is intended. The section "Number or String?" contains more information about this.) Thus, the pattern `$3>100` selects lines where the third field exceeds 100, and the program

```
$1 >= "S"
```

selects lines that begin with the letters S through Z, namely,

USSR	8650	262	Asia
USA	3615	219	North America
Sudan	968	19	Africa

In the absence of any other information, **awk** treats fields as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file **countries** as input, prints the single line for which this test succeeds:

```
Australia 2968 14 Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

Regular Expressions

awk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called regular expressions, and are like those in **egrep**(1) and **lex**(1). The simplest regular expression is a string of characters enclosed in slashes, like

```
/Asia/
```

This program prints all input records that contain the substring

PATTERNS

Asia. (If a record contains Asia as part of a larger string like Asian or Pan-Asiatic, it is also printed.) In general, if *re* is a regular expression, then the pattern:

```
/re/
```

matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators `~` (matches) and `!~` (does not match). The program

```
$4 ~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field matches Asia, while the program

```
$4 !~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field does not match Asia.

In regular expressions, the symbols

```
\ ^ $ . [ ] * + ? ( ) |
```

are metacharacters with special meanings like the metacharacters in the UNIX shell. For example, the metacharacters `^` and `$` match the beginning and end, respectively, of a string, and the metacharacter `.` ("dot") matches any single character. Thus,

```
/^.$/
```

matches all records that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, `/[ABC]/` matches records containing any one of **A**, **B**, or **C** anywhere. Ranges of letters or digits can be abbreviated within brackets: `/[a-zA-Z]/` matches any single letter.

If the first character after the [is a ^, this complements the class so it matches any character not in the set: `/[^a-zA-Z]/` matches any non-letter. The program

```
$2 !~ /^[0-9]+$/
```

prints all records in which the second field is not a string of one or more digits (^ for beginning of string, [0-9]+ for one or more digits, and \$ for end of string). Programs of this nature are often used for data validation.

Parentheses () are used for grouping and the symbol | is used for alternatives. The program

```
/(apple|cherry) (pie|tart)/
```

matches lines containing any one of the four substrings apple pie, apple tart, cherry pie, or cherry tart .

To turn off the special meaning of a metacharacter, precede it by a \ (backslash). Thus, the program

```
/b\$/
```

prints all lines containing b followed by a dollar sign.

In addition to recognizing metacharacters, the **awk** command recognizes the following C programming language escape sequences within regular expressions and strings:

- `\b` backspace
- `\f` formfeed
- `\n` newline
- `\r` carriage return
- `\t` tab
- `\ddd` octal value *ddd*
- `\"` quotation mark
- `\c` any other character *c* literally

201

PATTERNS

For example, to print all lines containing a tab, use the program

```
/\t/
```

awk interprets any string or variable on the right side of a `~` or `!~` as a regular expression. For example, we could have written the program

```
$2 !~ /^[0-9]+$ /
```

as

```
BEGIN { digits = "[0-9]+$" }
$2 !~ digits
```

Suppose you wanted to search for a string of characters like `^[0-9]+$`. When a literal quoted string like `"^[0-9]+$"` is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing `b` followed by a dollar sign. The regular expression for this pattern is `b\$`. If we want to create a string to represent this regular expression, we must add one more backslash: `"b\\$"`. The two regular expressions on each of the following lines are equivalent:

<code>x ~ "b\\\$"</code>	<code>x ~ /b\\$/</code>
<code>x ~ "b\\$"</code>	<code>x ~ /b\$/</code>
<code>x ~ "b\$"</code>	<code>x ~ /b\$/</code>
<code>x ~ "\\t"</code>	<code>x ~ /\t/</code>

The precise form of regular expressions and the substrings they match is given in Figure 4-4. The unary operators `*`, `+`, and `?` have the highest precedence, then concatenation, and then alternation `|`. All operators are left associative. `r` stands for any regular expression.

202

Expression	Matches
<i>c</i>	any non-metacharacter <i>c</i>
<i>\c</i>	character <i>c</i> literally
<i>^</i>	beginning of string
<i>\$</i>	end of string
<i>.</i>	any character but newline
<i>[s]</i>	any character in set <i>s</i>
<i>[^s]</i>	any character not in set <i>s</i>
<i>r*</i>	zero or more <i>r</i> 's
<i>r+</i>	one or more <i>r</i> 's
<i>r?</i>	zero or one <i>r</i>
<i>(r)</i>	<i>r</i>
<i>r₁r₂</i>	<i>r₁</i> then <i>r₂</i> (concatenation)
<i>r₁ r₂</i>	<i>r₁</i> or <i>r₂</i> (alternation)

Figure 4-4: **awk** Regular Expressions

203

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators **||** (or), **&&** (and), and **!** (not). For example, suppose we want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is *Asia* and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The program

```
$4 == "Asia" || $4 == "Africa"
```

PATTERNS

selects lines with Asia or Africa as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator | :

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator ! has the highest precedence, then &&, and finally ||. The operators && and || evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in

```
pat1, pat2 { ... }
```

In this case, the action is performed for each line between an occurrence of pat₁ and the next occurrence of pat₂ (inclusive). As an example, the pattern

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string Canada up through the next occurrence of the string Brazil:

Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America

Similarly, since FNR is the number of the current record in the current input file (and FILENAME is the name of the current input file), the program

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

prints the first five records of each input file with the name of the current input file prepended.

ACTIONS

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in Variables

Figure 4-5 lists the built-in variables that **awk** maintains. Some of these we have already met; others are used in this and later sections.

Variable	Meaning	Default
ARGC	number of command-line arguments	—
ARGV	array of command-line arguments	—
FILENAME	name of current input file	—
FNR	record number in current file	—
FS	input field separator	blank&tab
NF	number of fields in current record	—
NR	number of records read so far	—
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first char. matched by match()	—
RLENGTH	length of string matched by match()	—
SUBSEP	subscript separator	"\034"

Figure 4-5: **awk** Built-in Variables

ACTIONS

Arithmetic

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file **countries**. Since the second field is the area in thousands of square miles and the third field is the population in millions, the expression **1000 * \$3 / \$2** gives the population density in people per square mile. The program

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

applied to the file **countries** prints the name of each country and its population density:

USSR	30.3
Canada	6.2
China	234.6
USA	60.6
Brazil	35.3
Australia	4.7
India	502.0
Argentina	24.3
Sudan	19.6
Algeria	19.6

Arithmetic is done internally in floating point. The arithmetic operators are **+**, **-**, *****, **/**, **%** (remainder) and **^** (exponentiation; ****** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **awk** recognizes and produces scientific (exponential) notation: **1e6**, **1E6**, **10e5**, and **1000000** are numerically equal.

awk has assignment statements like those found in the C programming language. The simplest form is the assignment statement

$$v = e$$

where *v* is a variable or field name, and *e* is an expression. For example, to compute the number of Asian countries and their total

population, we could write

```
$4 == "Asia" { pop = pop + $3; n = n + 1 }
END          { print "population of", n,
               "Asian countries in millions is", pop }
```

Applied to **countries**, this program produces

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == "Asia"` contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **awk** initializes each variable with the string value "" and the numeric value 0.

The assignments in the previous program can be written more concisely using the operators `+=` and `++`:

```
$4 == "Asia" { pop += $3; ++n }
```

The operator `+=` is borrowed from the C programming language:

```
pop += $3
```

has the same effect as

```
pop = pop + $3
```

but the `+=` operator is shorter and runs faster. The same is true of the `++` operator, which adds one to a variable.

The abbreviated assignment operators are `+=`, `-=`, `*=`, `/=`, `%=`, and `^=`. Their meanings are similar:

```
v op = e
```

has the same effect as

```
v = v op e.
```

ACTIONS

The increment operators are `++` and `--`. As in C, they may be used as prefix (`++x`) or postfix (`x++`) operators. If `x` is 1, then `i=++x` increments `x`, then sets `i` to 2, while `i=x++` sets `i` to 1, then increments `x`. An analogous interpretation applies to prefix and postfix `--`.

Assignment and increment and decrement operators may all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3 { maxpop = $3; country = $1 }
END          { print country, maxpop }
```

Note, however, that this program would not be correct if all values of `$3` were negative.

awk provides the built-in arithmetic functions shown in Figure 4-6.

Function	Value Returned
atan2 (<i>y,x</i>)	arctangent of <i>y/x</i> in the range $-\pi$ to π
cos (<i>x</i>)	cosine of <i>x</i> , with <i>x</i> in radians
exp (<i>x</i>)	exponential function of <i>x</i>
int (<i>x</i>)	integer part of <i>x</i> truncated towards 0
log (<i>x</i>)	natural logarithm of <i>x</i>
rand ()	random number between 0 and 1
sin (<i>x</i>)	sine of <i>x</i> , with <i>x</i> in radians
sqrt (<i>x</i>)	square root of <i>x</i>
srand (<i>x</i>)	<i>x</i> is new seed for rand ()

Figure 4-6: **awk** Built-in Arithmetic Functions

x and *y* are arbitrary expressions. The function **rand**() returns a pseudo-random floating point number in the range (0,1), and **srand**(*x*) can be used to set the seed of the generator. If **srand**() has no argument, the seed is derived from the time of day.

Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in "abc" or "hello, everyone". String constants may contain the C programming language escape sequences for special characters listed in "Regular Expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions.

The program

```
{ print NR ":" $0 }
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

awk provides the built-in string functions shown in Figure 4-7. In this table, *r* represents a regular expression (either as a string or as /*r*/), *s* and *t* string expressions, and *n* and *p* integers.

Function	Description
gsub (<i>r,s</i>)	substitute <i>s</i> for <i>r</i> globally in current record, return number of substitutions
gsub (<i>r,s,t</i>)	substitute <i>s</i> for <i>r</i> globally in string <i>t</i> , return number of substitutions
index (<i>s,t</i>)	return position of string <i>t</i> in <i>s</i> , 0 if not present
length (<i>s</i>)	return length of <i>s</i>
match (<i>s,r</i>)	return the position in <i>s</i> where <i>r</i> occurs, 0 if not present
split (<i>s,a</i>)	split <i>s</i> into array <i>a</i> on FS, return number of fields

ACTIONS

Function	Description
split (<i>s,a,r</i>)	split <i>s</i> into array <i>a</i> on <i>r</i> , return number of fields
sprintf (<i>fmt,expr-list</i>)	return <i>expr-list</i> formatted according to format string <i>fmt</i>
sub (<i>r,s</i>)	substitute <i>s</i> for first <i>r</i> in current record, return number of substitutions
sub (<i>r,s,t</i>)	substitute <i>s</i> for first <i>r</i> in <i>t</i> , return number of substitutions
substr (<i>s,p</i>)	return suffix of <i>s</i> starting at position <i>p</i>
substr (<i>s,p,n</i>)	return substring of <i>s</i> of length <i>n</i> starting at position <i>p</i>

Figure 4-7: **awk** Built-in String Functions

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(1). The function **gsub**(*r,s,t*) replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in **ed**, the leftmost match is used, and is made as long as possible.) It returns the number of substitutions made. The function **gsub**(*r,s*) is a synonym for **gsub**(*r,s,\$0*). For example, the program

```
{ gsub(/USA/, "United States"); print }
```

transcribes its input, replacing occurrences of USA by United States. The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function **index**(*s,t*) returns the leftmost position where the string *t* begins in *s*, or zero if *t* does not occur in *s*. The first character in a string is at position 1. For example,

```
index("banana", "an")
```

returns 2.

The **length** function returns the number of characters in its argument string; thus,

```
{ print length($0), $0 }
```

prints each record, preceded by its length. (**\$0** does not include the input record separator.) The program

```
length($1) > max { max = length($1); name = $1 }
END              { print name }
```

applied to the file **countries** prints the longest country name: Australia.

The **match(s,r)** function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function also sets two built-in variables **RSTART** and **RLENGTH**. **RSTART** is set to the starting position of the match in the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. (If a match does not occur, **RSTART** is 0, and **RLENGTH** is -1.) For example, the following program finds the first occurrence of the letter *i* followed by at most one character followed by the letter *a* in a record:

```
{ if (match($0, /i.?a/))
  print RSTART, RLENGTH, $0 }
```

It produces the following output on the file **countries**:

17	2	USSR	8650	262	Asia
26	3	Canada	3852	24	North America
3	3	China	3692	866	Asia
24	3	USA	3615	219	North America
27	3	Brazil	3286	116	South America
8	2	Australia	2968	14	Australia
4	2	India	1269	637	Asia
7	3	Argentina	1072	26	South America
17	3	Sudan	968	19	Africa
6	2	Algeria	920	18	Africa

ACTIONS

NOTE

match() matches the left-most longest matching string. For example, with the record

```
AsiaaaAsiaaaaaan
```

as input, the program

```
{ if (match($0, /a+/)) print RSTART, RLENGTH, $0 }
```

matches the first string of a's and sets RSTART to 4 and RLENGTH to 3.

The function **sprintf**(*format*, *expr*₁, *expr*₂, . . . , *expr*_{*n*}) returns (without printing) a string containing *expr*₁, *expr*₂, . . . , *expr*_{*n*} formatted according to the **printf** specifications in the string *format*. "The **printf** Statement" in this chapter contains a complete specification of the format conventions. The statement

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to *x* the string produced by formatting the values of \$1 and \$2 as a ten-character string and a decimal number in a field of width at least six; *x* may be used in any subsequent computation.

The function **substr**(*s,p,n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr**(*s,p*) is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in **countries** to their first three characters by invoking the program

```
{ $1 = substr($1, 1, 3); print }
```

on this file to produce

```

USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa

```

Note that setting \$1 in the program forces **awk** to recompute \$0 and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on file **countries**,

```

    { s = s substr($1, 1, 3) " " }
END { print s }

```

prints

```

USS Can Chi USA Bra Aus Ind Arg Sud Alg

```

by building **s** up a piece at a time from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables **\$1**, **\$2**, ..., **\$NF**. Field variables share all of the properties of other variables — they may be used in arithmetic or string operations, and they may have values assigned to them. So, for example, you can divide the second field of the file **countries** by 1000 to convert the area from thousands to millions of square miles:

```

{ $2 /= 1000; print }

```

ACTIONS

or assign a new string to a field:

```
BEGIN                { FS = OFS = "\t" }
$4 == "North America" { $4 = "NA" }
$4 == "South America" { $4 = "SA" }
                    { print }
```

The `BEGIN` action in this program resets the input field separator `FS` and the output field separator `OFS` to a tab. Notice that the `print` in the fourth line of the program prints the value of `$0` after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, `$(NF-1)` is the second to last field of the current record. The parentheses are needed: the value of `$(NF-1)` is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, `$(NF+1)`, has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file `countries` creates a fifth field giving the population density:

```
BEGIN { FS = OFS = "\t" }
      { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

Number or String?

Variables, fields and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

`pop` and `$3` must be treated numerically, so their values will be coerced to numeric type if necessary.

In a string context like

```
print $1 ":" $2
```

\$1 and \$2 must be strings to be concatenated, so they will be coerced if necessary.

In an assignment $v = e$ or $v\ op = e$, the type of v becomes the type of e . In an ambiguous context like

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison "\$1 == \$2" will succeed on any pair of the inputs

```
1      1.0    +1     0.1e+1    10E-1    001
```

but fail on the inputs

```
(null) 0
(null) 0.0
0a      0
1e50    1.0e50
```

There are two idioms for coercing an expression of one type to the other:

<i>number</i> ""	concatenate a null string to a <i>number</i> to coerce it to type string
<i>string</i> + 0	add zero to a <i>string</i> to coerce it to type numeric

ACTIONS

Thus, to force a string comparison between two fields, say

```
$1 "" = = $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**.

Uninitialized variables have numeric value 0 and string value **""**. Nonexistent fields and fields that are explicitly null have only the string value **""**; they are not numeric.

Control Flow Statements

awk provides **if-else**, **while**, **do-while**, and **for** statements, and statement grouping with braces, as in the C programming language.

The **if** statement syntax is

```
if (expression) statement1 else statement2
```

The *expression* acting as the conditional has no restrictions; it can include the relational operators **<**, **<=**, **>**, **>=**, **==**, and **!=**; the regular expression matching operators **~** and **!~**; the logical operators **||**, **&&**, and **!**; juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, the *expression* is first evaluated. If it is non-zero and non-null, *statement*₁ is executed; otherwise *statement*₂ is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from "Arithmetic Functions" with an **if** statement results in

```

    {   if (maxpop < $3) {
            maxpop = $3
            country = $1
        }
    }
END { print country, maxpop }

```

The **while** statement is exactly that of the C programming language:

while (*expression*) *statement*

The *expression* is evaluated; if it is non-zero and non-null the *statement* is executed and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, to print all input fields one per line,

```

    {   i = 1
        while (i <= NF) {
            print $i
            i++
        }
    }

```

The **for** statement is like that of the C programming language:

for (*expression*₁; *expression*; *expression*₂) *statement*

It has the same effect as

```

expression1
while (expression) {
    statement
    expression2
}

```

ACTIONS

so

```
{ for (i = 1; i <= NF; i++) print $i }
```

does the same job as the **while** example above. An alternate version of the **for** statement is described in the next section.

The **do** statement has the form

```
do statement while (expression)
```

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement* (at the bottom of the loop), it is always executed at least once. As a result, the **do** statement is used much less often than **while** or **for**, which test for completion at the top of the loop.

The following example of a **do** statement prints all lines except those between **start** and **stop**.

```
/start/ {
    do {
        getline x
    } while (x !~ /stop/)
}
{ print }
```

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **awk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the **END** action, if any, is executed. Within the **END** action,

```
exit expr
```

causes the program to return the value of *expr* as its exit status. If there is no *expr*, the exit status is zero.

Arrays

awk provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they spring into existence by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array *x*. In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **awk** program

```
{ x[NR] = $0 }
END { ... processing ... }
```

The first action merely records each input line in the array *x*, indexed by line number; processing is done in the **END** statement.

Array elements may also be named by nonnumeric values. For example, the following program accumulates the total population of Asia and Africa into the associative array *pop*. The **END** action prints the total population of these two continents.

```
/Asia/    { pop["Asia"] += $3 }
/Africa/  { pop["Africa"] += $3 }
END       { print "Asian population in millions is",
            pop["Asia"]
            print "African population in millions is",
            pop["Africa"] }
```

On the file **countries**, this program generates

```
Asian population in millions is 1765
African population in millions is 37
```

In this program if we had used `pop[Asia]` instead of `pop["Asia"]` the expression would have used the value of the variable *Asia* as the subscript, and since the variable is uninitialized, the values would have been accumulated in `pop[""]`.

ACTIONS

Suppose our task is to determine the total area in each continent of the file **countries**. Any expression can be used as a subscript in an array reference. Thus

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array `area` and in that entry accumulates the value of the second field:

```
BEGIN { FS = "\t" }
        { area[$4] += $2 }
END    { for (name in area)
          print name, area[name] }
```

Invoked on the file **countries**, this program produces

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

This program uses a form of the **for** statement that iterates over all defined subscripts of an array:

for (*i in array*) *statement*

executes *statement* with the variable *i* set in turn to each value of *i* for which *array[i]* has been defined. The loop is executed once for each defined subscript, which are chosen in a random order. Results are unpredictable when *i* or *array* is altered during the loop.

awk does not provide multi-dimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable **SUBSEP**). For example,

```

for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
        arr[i,j] = ...

```

creates an array which behaves like a two-dimensional array; the subscript is the concatenation of *i*, **SUBSEP**, and *j*.

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr*, as in

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating `area["Africa"]`, which would happen if we used

```
if (area["Africa"] != "") ...
```

Note that neither is a test of whether the array `area` contains an element with value "Africa" .

It is also possible to split any string into fields in the elements of an array using the built-in function **split**. The function

```
split("s1:s2:s3", a, ":")
```

splits the string `s1:s2:s3` into three fields, using the separator `:` , and stores `s1` in `a[1]`, `s2` in `a[2]`, and `s3` in `a[3]` . The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element may be deleted with the **delete** statement:

```
delete arrayname[subscript]
```

ACTIONS

User-Defined Functions

awk provides user-defined functions. A function is defined as

```
function name(argument-list) {  
    statements  
}
```

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function these variables refer to the actual parameters when the function is called. There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, the following program defines and tests the usual recursive factorial function (of course, using some input other than the file **countries**):

```
function fact(n) {  
    if (n <= 1)  
        return 1  
    else  
        return n * fact(n-1)  
}  
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however, so the function cannot affect their values outside. Within a function, formal parameters are local variables but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables.) The **return** statement is optional, but the returned value is undefined if it is not included.

Some Lexical Conventions

Comments may be placed in **awk** programs: they begin with the character **#** and end at the end of the line, as in

```
print x, y    # this is a comment
```

Statements in an **awk** program normally occupy a single line. Several statements may occur on a single line if they are separated by semicolons. A long statement may be continued over several lines by terminating each continued line by a backslash. (It is not possible to continue a "... " string.) This explicit continuation is rarely necessary, however, since statements continue automatically if the line ends with a comma (for example, as might occur in a **print** or **printf** statement) or after the operators **&&** and **||**.

Several pattern-action statements may appear on a single line if separated by semicolons.

This page is intentionally left blank

OUTPUT

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **awk** lets you redirect output, so that output from **print** and **printf** can be directed to files and pipes. This section describes the use of these two statements.

The print Statement

The statement

```
print expr1, expr2, . . . , exprn
```

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement

```
print
```

is an abbreviation for

```
print $0
```

To print an empty line use

```
print ""
```

Output Separators

The output field separator and record separator are held in the built-in variables **OFS** and **ORS**. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines

OUTPUT

after the second field:

```
BEGIN { OFS = ":"; ORS = "\n\n" }
      { print $1, $2 }
```

Notice that

```
{ print $1 $2 }
```

prints the first and second fields with no intervening output field separator, because `$1 $2` is a string consisting of the concatenation of the first two fields.

The printf Statement

awk's printf statement is the same as that in C except that the `*` format specifier is not supported. The **printf** statement has the general form

```
printf format, expr1, expr2, . . . , exprn
```

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Figure 4-8. Each specification begins with a `%`, ends with a letter that determines the conversion, and may include

- left-justify expression in its field
- width* pad field to this width as needed; fields that begin with a leading 0 are padded with zeros
- .prec* maximum string width or digits to right of decimal point

Character	Prints Expression as
c	single character
d	decimal number
e	<code>[-]d.dddddE[+ -]dd</code>
f	<code>[-]ddd.ddddd</code>
g	e or f conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a % ; no argument is converted

Figure 4-8: **awk printf** Conversion Characters

Here are some examples of **printf** statements along with the corresponding output:

```

printf "%d", 99/2           49
printf "%e", 99/2         4.950000e+01
printf "%f", 99/2         49.500000
printf "%6.2f", 99/2      49.50
printf "%g", 99/2         49.5
printf "%o", 99           143
printf "%06o", 99         000143
printf "%x", 99           63
printf "|%s|", "January"  |January|
printf "|%10s|", "January" |   January|
printf "|%-10s|", "January" |January |
printf "|%.3s|", "January" |Jan|
printf "|%10.3s|", "January" |           Jan|
printf "|%-10.3s|", "January" |Jan      |
printf "%%"               %
    
```

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

OUTPUT

Output into Files

It is possible to print output into files instead of to the standard output by using the `>` and `>>` redirection operators. For example, the following program invoked on the file **countries** prints all lines where the population (third field) is bigger than 100 into a file called **bigpop**, and all other lines into **smallpop**:

```
$3 > 100 { print $1, $3 >"bigpop" }
$3 <= 100 { print $1, $3 >"smallpop" }
```

Notice that the file names have to be quoted; without quotes, **bigpop** and **smallpop** are merely uninitialized variables. If the output file names were created by an expression, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME) }
```

This is because the `>` operator has higher precedence than concatenation; without parentheses, the concatenation of **tmp** and **FILENAME** would not work.

NOTE

Files are opened once in an **awk** program. If `>` is used to open a file, its original contents are overwritten. But if `>>` is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.

Output into Pipes

It is also possible to direct printing into a pipe with a command on the other end, instead of into a file. The statement

```
print | "command-line"
```

causes the output of **print** to be piped into the *command-line*.

Although we have shown them here as literal strings enclosed in quotes, the *command-line* and file names can come from variables and the return values from functions, for instance.

Suppose we want to create a list of continent-population pairs, sorted alphabetically by continent. The **awk** program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called `pop`. Then it prints each continent and its population, and pipes this output into the `sort` command.

```
BEGIN    { FS = "\t" }
          { pop[$4] += $3 }
END      { for (c in pop)
          print c ":" pop[c] | "sort" }
```

Invoked on the file **countries**, this program yields

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these **print** statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named `sort`), but they are created and opened only once in the entire run. So, in the last example, for all `c` in `pop`, only one `sort` pipe is open.

There is a limit to the number of files that can be open simultaneously. The statement **close**(*file*) closes a file or pipe; *file* is the string used to create it in the first place, as in

```
close("sort")
```

When opening or closing a file, different strings are different commands.

This page is intentionally left blank

INPUT

The most common way to give input to an **awk** program is to name on the command line the file(s) that contains the input. This is the method we've been using in this chapter. However, there are several other methods we could use, each of which this section describes.

Files and Pipes

You can provide input to an **awk** program by putting the input data into a file, say **awkdata**, and then executing

```
awk 'program' awkdata
```

awk reads its standard input if no file names are given (see "Usage" in this chapter); thus, a second common arrangement is to have another program pipe its output into **awk**. For example, **egrep**(1) selects input lines containing a specified regular expression, but it can do so faster than **awk** since this is the only thing it does. We could, therefore, invoke the pipe

```
egrep 'Asia' countries | awk '...'
```

egrep quickly finds the lines containing **Asia** and passes them on to the **awk** program for subsequent processing.

Input Separators

With the default setting of the field separator **FS**, input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
    field1  field2
  field1
field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS**. For example,

```
BEGIN { FS = "(,[ \\t]*)|([ \\t]+)" }
```

sets it to an optional comma followed by any number of blanks and tabs. **FS** can also be set on the command line with the **-F** argument:

```
awk -F'([ \\t]*)|([ \\t]+)''...'
```

behaves the same as the previous example. Regular expressions used as field separators match the left-most longest occurrences (as in **sub()**), but do not match null strings.

Multi-line Records

Records are normally separated by newlines, so that each line is a record, but this too can be changed, though only in a limited way. If the built-in record separator variable **RS** is set to the empty string, as in

```
BEGIN { RS = "" }
```

then input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line

records is to use

```
BEGIN { RS = ""; FS = "\n" }
```

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. "The **getline** Function" and "Cooperation with the Shell" in this chapter show other examples of processing multi-line records.

The **getline** Function

awk's facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting **RS** to null doesn't work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

The function **getline** can be used to read input either from the current input or from a file or pipe, by redirection analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF**, **NR**, and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and -1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with **START** and ends with a line beginning with **STOP**. The following **awk** program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array

```
f[1] f[2] ... f[nf]
```

Once the line containing **STOP** is encountered, the record can be processed from the data in the **f** array:

INPUT

```

/^START/ {
    f[nf=1] = $0
    while (getline && $0 !~ /^STOP/)
        f[++nf] = $0
    # now process the data in f[1]...f[nf]
    ...
}

```

Notice that this code uses the fact that **&&** evaluates its operands left to right and stops as soon as one is true.

The same job can also be done by the following program:

```

/^START/ && nf= =0 { f[nf=1] = $0 }
nf > 1 { f[++nf] = $0 }
/^STOP/ { # now process the data in f[1]...f[nf]
    ...
    nf = 0
}

```

The statement

```
getline x
```

reads the next record into the variable **x**. No splitting is done; **NF** is not set. The statement

```
getline <"file"
```

reads from **file** instead of the current input. It has no effect on **NR** or **FNR**, but field splitting is performed and **NF** is set. The statement

```
getline x <"file"
```

gets the next record from **file** into **x**; no splitting is done, and **NF**, **NR** and **FNR** are untouched.

NOTE

If a filename is an expression, it needs to be placed in parentheses for correct evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) { ... }
```

This is because the `<` has precedence over concatenation. Without parentheses, a statement such as

```
getline x < "tmp" FILENAME
```

sets `x` to read the file `tmp` and not `tmp <value of FILENAME>`.

Also, if you use this **getline** statement form, a statement like

```
while ( getline x < file ) { ... }
```

loops forever if the file cannot be read, because `getline` returns `-1`, not zero, if an error occurs. A better way to write this test is

```
while ( getline x < file > 0 ) { ... }
```

It is also possible to pipe the output of another command directly into **getline**. For example, the statement

```
while ("who" | getline)
    n++
```

executes `who` and pipes its output into `getline`. Each iteration of the while loop reads one more line and increments the variable `n`, so after the while loop terminates, `n` contains a count of the number of users. Similarly, the statement

```
"date" | getline d
```

pipes the output of `date` into the variable `d`, thus setting `d` to the current date. Figure 4-9 summarizes the **getline** function.

INPUT

Form	Sets
<code>getline</code>	<code>\$0, NF, NR, FNR</code>
<code>getline var</code>	<code>var, NR, FNR</code>
<code>getline <file</code>	<code>\$0, NF</code>
<code>getline var <file</code>	<code>var</code>
<code>cmd getline</code>	<code>\$0, NF</code>
<code>cmd getline var</code>	<code>var</code>

Figure 4-9: `getline` Function

Command-line Arguments

The command-line arguments are available to an `awk` program: the array `ARGV` contains the elements `ARGV[0]`, ..., `ARGV[ARGC - 1]`; as in C, `ARGC` is the count. `ARGV[0]` is the name of the program (generally `awk`); the remaining arguments are whatever was provided (excluding the program and any optional arguments).

The following command line contains an `awk` program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
    for (i = 1; i < ARGC; i++)
        printf "%s ", ARGV[i]
    printf "\n"
}' $ *
```

The arguments may be modified or added to; `ARGC` may be altered. As each input file ends, `awk` treats the next non-null element of `ARGV` (up to the current value of `ARGC - 1`) as the name of the next input file.

There is one exception to the rule that an argument is a file name: if it is of the form

var=value

then the variable *var* is set to the value *value* as if by assignment. Such an argument is not treated as a file name. If *value* is a string, no quotes are needed.

INPUT

This page is intentionally left blank

Using `awk` with Other Commands and the Shell

`awk` gains its greatest power when it is used in conjunction with other programs. Here we describe some of the ways in which `awk` programs cooperate with other commands.

The `system` Function

The built-in function `system(command-line)` executes the command *command-line*, which may well be a string computed by, for example, the built-in function `sprintf`. The value returned by `system` is the return status of the command executed.

For example, the program

```
$1 = = "#include" { gsub(/[<>"]/, "", $2);
                    system("cat " $2) }
```

calls the command `cat` to print the file named in the second field of every input record whose first field is `#include`, after stripping any `<`, `>` or `"` that might be present.

Cooperation with the Shell

In all the examples thus far, the `awk` program was in a file and fetched from there using the `-f` flag, or it appeared on the command line enclosed in single quotes, as in

```
awk '{ print $1 }' ...
```

Since `awk` uses many of the same characters as the shell does, such as `$` and `"`, surrounding the `awk` program with single quotes ensures that the shell will pass the entire program unchanged to the `awk` interpreter.

Using awk with Other Commands and the Shell

Now, consider writing a command **addr** that will search a file **addresslist** for name, address and telephone information. Suppose that **addresslist** contains names and addresses in which a typical entry is a multi-line record such as

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

We want to search the address list by issuing commands like

```
addr Emlin
```

That is easily done by a program of the form

```
awk '
BEGIN { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called **addr** that contains

```
awk '
BEGIN { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here: the **awk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The **\$1** is outside the quotes, visible to the shell, which therefore replaces it by the pattern **Emlin** when the command **addr Emlin** is invoked. On a UNIX system, **addr** can be made executable by changing its mode with the following command: **chmod +x addr**.

Using **awk** with Other Commands and the Shell

A second way to implement **addr** relies on the fact that the shell substitutes for **\$** parameters within double quotes:

```
awk "
BEGIN { RS = "\" }
/$1/
" addresslist
```

Here we must protect the quotes defining **RS** with backslashes, so that the shell passes them on to **awk**, uninterpreted by the shell. **\$1** is recognized as a parameter, however, so the shell replaces it by the pattern when the command **addr pattern** is invoked.

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **awk** program that explicitly reads through the address list with **getline**:

```
awk '
BEGIN { RS = ""
        while (getline < "addresslist")
            if ($0 ~ ARGV[1])
                print $0
    } ' $*
```

All processing is done in the **BEGIN** action.

Notice that any regular expression can be passed to **addr**; in particular, it is possible to retrieve by parts of an address or telephone number as well as by name.

Using awk with Other Commands and the Shell

This page is intentionally left blank

Example Applications

awk has been used in surprising ways. We have seen **awk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. In this section, we will present a few more examples to illustrate some additional **awk** programs.

Generating Reports

awk is especially useful for producing reports that summarize and format information. Suppose we wish to produce a report from the file **countries** in which we list the continents alphabetically, and after each continent its countries in decreasing order of population:

Africa:	Sudan	19
	Algeria	18
Asia:	China	866
	India	637
	USSR	262
Australia:	Australia	14
North America:	USA	219
	Canada	24
South America:	Brazil	116
	Argentina	26

As with many data processing tasks, it is much easier to produce this report in several stages. First, we create a list of continent-country-population triples, in which each field is separated by a colon. This

Example Applications

can be done with the following program **triples**, which uses an array **pop** indexed by subscripts of the form 'continent:country' to store the population of a given country. The print statement in the **END** section of the program creates the list of continent-country-population triples that are piped to the **sort** routine.

```
BEGIN { FS = "\t" }
        { pop[$4 ":" $1] += $3 }
END    { for (cc in pop)
        print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for **sort** deserve special mention. The **-t:** argument tells **sort** to use **:** as its field separator. The **+0 -1** arguments make the first field the primary sort key. In general, **+i -j** makes fields **i+1, i+2, ..., j** the sort key. If **-j** is omitted, the fields from **i+1** to the end of the record are used. The **+2nr** argument makes the third field, numerically decreasing, the secondary sort key (**n** is for numeric, **r** for reverse order). Invoked on the file **countries**, this program produces as output

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form we run it through a second **awk** program **format**:

```

BEGIN { FS = ":" }
{
    if ($1 != prev) {
        print "\n" $1 ":"
        prev = $1
    }
    printf "\t%-10s %6d\n", $2, $3
}

```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line

awk -f triples countries | awk -f format

gives us our desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **awks** and **sorts**.

As an exercise, add to the population report subtotals for each continent and a grand total.

Additional Examples

Word Frequencies

Our first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order.

```

{ for (w = 1; w <= NF; w++) count[$w]++ }
END { for (w in count) print count[w], w | "sort -nr" }

```

The first statement uses the array `count` to accumulate the number of times each word is used. Once the input has been read, the second `for` loop pipes the final count along with each word into the `sort` command.

Example Applications

Accumulation

Suppose we have two files, `deposits` and `withdrawals`, of records containing a name field and an amount field. For each name we want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
awk '
FILENAME == "deposits"    { balance[$1] += $2 }
FILENAME == "withdrawals" { balance[$1] -= $2 }
END                       { for (name in balance)
                           print name, balance[name]
                           }
}' deposits withdrawals
```

The first statement uses the array `balance` to accumulate the total amount for each name in the file `deposits`. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name will be created by the second statement. The `END` action prints each name with its net balance.

Random Choice

The following function prints (in order) k random elements from the first n elements of the array `A`. In the program, k is the number of entries that still need to be printed, and n is the number of elements yet to be examined. The decision of whether to print the i th element is determined by the test `rand() < k/n`.

```
function choose(A, k, n) {
    for (i = 1; n > 0; i++) {
        if (rand() < k/n--) {
            print A[i]
            k--
        }
    }
}
```

Shell Facility

The following **awk** program simulates (crudely) the history facility of the UNIX system shell. A line containing only `=` re-executes the last command executed. A line beginning with `= cmd` re-executes the last command whose invocation included the string `cmd`. Otherwise, the current line is executed.

```

$1 == "=" { if (NF == 1)
              system(x[NR] = x[NR-1])
            else
              for (i = NR-1; i > 0; i--)
                if (x[i] ~ $2) {
                    system(x[NR] = x[i])
                    break
                }
              next }

././        { system(x[NR] = $0) }

```

247

Form-letter Generation

The following program generates form letters, using a template stored in a file called `form.letter`:

```

This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.

```

and replacement text of this form:

```

field 1|field 2|field 3
one|two|three
a|b|c

```

The `BEGIN` action stores the template in the array `template`; the remaining action cycles through the input data, using `gsub` to replace

Example Applications

template fields of the form $\$n$ with the corresponding data fields.

```
BEGIN {      FS = "|"
            while (getline <"form.letter")
                line[++n] = $0
        }
        {    for (i = 1; i <= n; i++) {
                s = line[i]
                for (j = 1; j <= NF; j++)
                    gsub("\\\\\$"j, $j, s)
                print s
            }
        }
```

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

awk Summary

Command Line

awk *program filenames*

awk -f *program-file filenames*

awk -Fs sets field separator to string *s*; **-Ft** sets separator to tab

Patterns

BEGIN

END

/regular expression/

relational expression

pattern && pattern

pattern || pattern

(pattern)

!pattern

pattern, pattern

Control Flow Statements

if (*expr*) *statement* [**else** *statement*]

if (*subscript in array*) *statement* [**else** *statement*]

while (*expr*) *statement*

for (*expr*; *expr*; *expr*) *statement*

for (*var in array*) *statement*

do *statement* **while** (*expr*)

break

continue

next

exit [*expr*]

return [*expr*]

awk Summary

Input-output

close (<i>filename</i>)	close file
getline	set \$0 from next input record; set NF , NR
getline < <i>file</i>	set \$0 from next record of <i>file</i> ; set NF
getline <i>var</i>	set <i>var</i> from next input record; set NR , FNR
getline <i>var</i> < <i>file</i>	set <i>var</i> from next record of <i>file</i>
print	print current record
print <i>expr-list</i>	print expressions
print <i>expr-list</i> > <i>file</i>	print expressions on <i>file</i>
printf <i>fmt</i> , <i>expr-list</i>	format and print
printf <i>fmt</i> , <i>expr-list</i> > <i>file</i>	format and print on <i>file</i>
system (<i>cmd-line</i>)	execute command <i>cmd-line</i> , return status

In **print** and **printf** above, > >*file* appends to the *file*, and | "*command*" writes on a pipe. Similarly, "*command*" | **getline** pipes into **getline**. **getline** returns 0 on end of file, and -1 on error.

Functions

func *name*(*parameter list*) { *statement* }
function *name*(*parameter list*) { *statement* }
function-name(*expr*, *expr*, ...)

String Functions

gsub (<i>r,s,t</i>)	substitute string <i>s</i> for each substring matching regular expression <i>r</i> in string <i>t</i> , return number of substitutions; if <i>t</i> omitted, use \$0
index (<i>s,t</i>)	return index of string <i>t</i> in string <i>s</i> , or 0 if not present
length (<i>s</i>)	return length of string <i>s</i>
match (<i>s,r</i>)	return position in <i>s</i> where regular expression <i>r</i> occurs, or 0 if <i>r</i> is not present
split (<i>s,a,r</i>)	split string <i>s</i> into array <i>a</i> on regular expression <i>r</i> , return number of fields; if <i>r</i> omitted, FS is used in its place
sprintf (<i>fmt, expr-list</i>)	print <i>expr-list</i> according to <i>fmt</i> , return resulting string
sub (<i>r,s,t</i>)	like gsub except only the first matching substring is replaced
substr (<i>s,i,n</i>)	return <i>n</i> -char substring of <i>s</i> starting at <i>i</i> ; if <i>n</i> omitted, use rest of <i>s</i>

Arithmetic Functions

atan2 (<i>y,x</i>)	arctangent of <i>y/x</i> in radians
cos (<i>expr</i>)	cosine (angle in radians)
exp (<i>expr</i>)	exponential
int (<i>expr</i>)	truncate to integer
log (<i>expr</i>)	natural logarithm
rand ()	random number between 0 and 1
sin (<i>expr</i>)	sine (angle in radians)
sqrt (<i>expr</i>)	square root
srand (<i>expr</i>)	new seed for random number generator; use time of day if no <i>expr</i>

awk Summary

Operators (Increasing Precedence)

= += - = *= /= %= ^=	assignment
?:	conditional expression
	logical OR
&&	logical AND
~ !~	regular expression match, negated match
< <= > >= != ==	relationals
<i>blank</i>	string concatenation
+ -	add, subtract
* / %	multiply, divide, mod
+ - !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ --	increment, decrement (prefix and postfix)
\$	field

Regular Expressions (Increasing Precedence)

<i>c</i>	matches non-metacharacter <i>c</i>
<i>\c</i>	matches literal character <i>c</i>
.	matches any character but newline
^	matches beginning of line or string
\$	matches end of line or string
<i>[abc...]</i>	character class matches any of <i>abc...</i>
<i>[^abc...]</i>	negated class matches any but <i>abc...</i> and newline
<i>r1 r2</i>	matches either <i>r1</i> or <i>r2</i>
<i>r1r2</i>	concatenation: matches <i>r1</i> , then <i>r2</i>
<i>r+</i>	matches one or more <i>r</i> 's
<i>r*</i>	matches zero or more <i>r</i> 's
<i>r?</i>	matches zero or one <i>r</i> 's
<i>(r)</i>	grouping: matches <i>r</i>

Built-in Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0..ARGC-1)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default <code>% .6g</code>)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	index of first character matched by <code>match()</code> ; 0 if no match
RLENGTH	length of string matched by <code>match()</code> ; - 1 if no match
SUBSEP	separates multiple subscripts in array elements; default <code>"\034"</code>

Limits

Any particular implementation of **awk** enforces some limits. Here are typical values:

- 100 fields
- 2500 characters per input record
- 2500 characters per output record
- 1024 characters per individual field
- 1024 characters per printf string
- 400 characters maximum quoted string
- 400 characters in character class
- 15 open files
- 1 pipe
- numbers are limited to what can be represented on the local machine, e.g., `1e - 308..1e + 308`

awk Summary

Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment

```
var = expr
```

its type is set to that of the expression. (Assignment includes `+`, `-`, `=`, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of `v1` becomes that of `v2`.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

(that is, concatenation with a null string).

Uninitialized variables have the numeric value `0` and the string value `""`. Accordingly, if `x` is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
```

```
if (x == 0) ...
```

```
if (x == "") ...
```

are all true. But the following is false:

```
if (x == "0") ...
```

The type of a field is determined by context when possible; for example,

```
$1++
```

clearly implies that \$1 is to be numeric, and

```
$1 = $1 ", " $2
```

implies that \$1 and \$2 are both to be strings. Coercion is done as needed.

In contexts where types cannot be reliably determined, for example,

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value "" ; they are not numeric. Non-existent fields (i.e., fields past **NF**) are treated this way, too.

As it is for fields, so it is for array elements created by **split()**.

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus, if **arr[i]** does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value "" so the **if** is satisfied. The special construction

```
if (i in arr) ...
```

determines if **arr[i]** exists without the side effect of creating it if it does not.

This page is intentionally left blank

Chapter 5: lex

	Page
An Overview of lex Programming	5-1
Writing lex Programs	5-3
The Fundamentals of lex Rules.....	5-3
Specifications	5-3
Actions.....	5-6
Advanced lex Usage.....	5-8
Some Special Features.....	5-9
Definitions	5-13
Subroutines.....	5-15
Using lex with yacc	5-17
Running lex under the UNIX System.....	5-21

Table of Contents

This page is intentionally left blank

An Overview of `lex` Programming

`lex` is a software tool that lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you may check the spelling of words for errors; in code enciphering, you may translate certain patterns of characters into others; and in compiler writing, you may determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled. The problem common to all of these tasks is recognizing different strings of characters that satisfy certain characteristics. In the compiler writing case, creating the ability to solve the problem requires implementing the compiler's lexical analyzer. Hence the name `lex`.

It is not essential to use `lex` to handle problems of this kind. You could write programs in a standard language like C to handle them, too. In fact, what `lex` does is produce such C programs. (`lex` is therefore called a program generator.) What `lex` offers you is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In many applications this is a minor consideration, and the advantages of using `lex` considerably outweigh it.

To understand what `lex` does, see the diagram in Figure 5-1. We begin with the `lex` source (often called the `lex` specification) that you, the programmer, write to solve the problem at hand. This `lex` source consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. The source is read by the `lex` program generator. The output of the program generator is a C program that, in turn, must be compiled by a host language C compiler to generate the executable object program that does the lexical analysis. Note that this procedure is not typically automatic—user intervention is required. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

An Overview of `lex` Programming

`lex` can also be used to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, and so forth. In later sections of this chapter, we will see

- how to write `lex` source to do some of these tasks
- how to translate `lex` source
- how to compile, link, and execute the lexical analyzer in C
- how to run the lexical analyzer program

We will then be on our way to appreciating the power that `lex` provides.

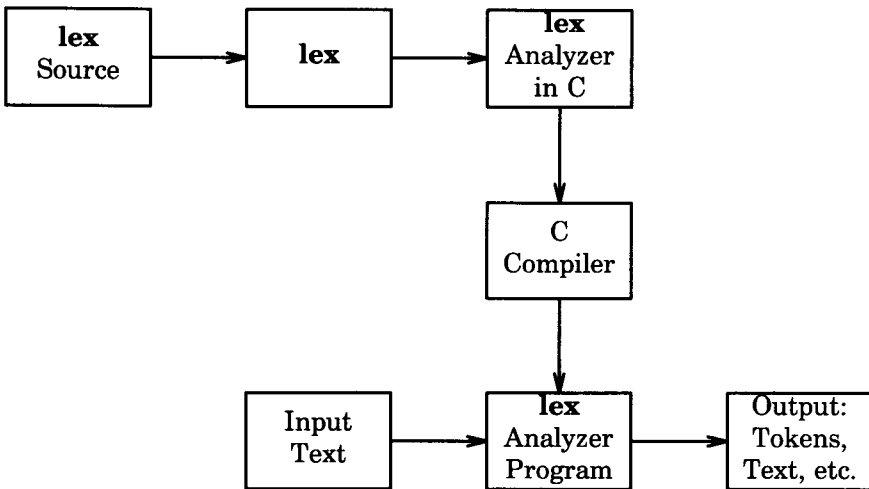


Figure 5-1: Creation and Use of a Lexical Analyzer with `lex`

Writing `lex` Programs

A `lex` specification consists of at most three sections: definitions, rules, and user subroutines. The rules section is mandatory. Sections for definitions and user subroutines are optional, but if present, must appear in the indicated order.

The Fundamentals of `lex` Rules

The mandatory rules section opens with the delimiter `%%`. If a subroutines section follows, another `%%` delimiter ends the rules section. If there is no second delimiter, the rules section is presumed to continue to the end of the program.

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. (Note the dual meaning of the term specification — it may mean either the entire `lex` source itself or, within it, a representation of a particular pattern to be recognized.) Whenever the input consists of patterns not sought, `lex` writes out the input exactly as it finds it. So, the simplest `lex` program is just the beginning rules delimiter, `%%`. It writes out the entire input to the output with no changes at all. Typically, the rules are more elaborate than that.

Specifications

You specify the patterns you are interested in with a notation called regular expressions. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all. For example,

```
apple
orange
pluto
```

These three regular expressions match any occurrences of those

Writing lex Programs

character strings in an input text. If you want to have your lexical analyzer **a.out** remove every occurrence of **orange**, from the input text, you could specify the rule

```
orange;
```

Because you did not specify an action on the right (before the semi-colon), **lex** does nothing but print out the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string **orange** at all.

Unlike **orange** above, most of the expressions that we want to search for cannot be specified so easily. The expression itself might simply be too long. More commonly, the class of desired expressions is too large; it may, in fact, be infinite. Thanks to the use of operators, we can form regular expressions signifying any expression of a certain class. The **+** operator, for instance, means one or more occurrences of the preceding expression, the **?** means 0 or 1 occurrence(s) of the preceding expression (this is equivalent, of course, to saying that the preceding expression is optional), and ***** means 0 or more occurrences of the preceding expression. (It may at first seem odd to speak of 0 occurrences of an expression and to need an operator to capture the idea, but it is often quite helpful. We will see an example in a moment.) So **m+** is a regular expression matching any string of **ms** such as each of the following:

```
mmm
m
mmmmm
mm
```

and **7*** is a regular expression matching any string of zero or more **7s**:

```
77
77777
777
```

The string of blanks on the third line matches simply because it has no **7s** in it at all.

Brackets, [], indicate any one character from the string of characters specified between the brackets. Thus, [dgka] matches a single **d**, **g**, **k**, or **a**. Note that commas are not included within the brackets. Any comma here would be taken as a character to be recognized in the input text. Ranges within a standard alphabetic or numeric order are indicated with a hyphen, -. The sequence [a-z], for instance, indicates any lowercase letter. Somewhat more interestingly:

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether upper- or lowercase), any digit, an asterisk, an ampersand, or a sharp character. Given the input text

```
$$$$?? ?????!!!*$$ $$$$$$&+= = = =r~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize the *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by a *, would match any digit or letter. The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the asterisk, *, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

Writing lex Programs

```
e
pay
distance
pH
EngineNo99
R2D2
```

Note that it would not recognize the following as identifiers:

```
not_ideNTIFER
5times
$hello
```

because **not_ideNTIFER** has an embedded underscore; **5times** starts with a digit, not a letter; and **\$hello** starts with a special character. Of course, you may want to write the specifications for these three examples as an exercise.

A potential problem with operator characters is how we can refer to them as characters to look for in our search pattern. The last example, for instance, will not recognize text with an `*` in it. **lex** solves the problem in one of two ways: a character enclosed in quotation marks or a character preceded by a `\` is taken literally, that is, as part of the text to be searched for. To use the backslash method to recognize, say, an `*` followed by any number of digits, we can use the pattern

```
\*[1-9]*
```

To recognize a `\` itself, we need two backslashes: `\\`.

Actions

Once **lex** recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. What you want to do is write these actions as program fragments in the host language C. An action may consist of as many statements as are needed for

the job at hand. You may want to print out a message noting that the text has been found or a message transforming the text in some way. Thus, to recognize the expression Amelia Earhart and to note such recognition, the rule

```
"Amelia Earhart"  printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent acronyms, a rule such as

```
Electroencephalogram  printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize end-of-lines and increment a linecounter. **lex** uses the standard escape sequences from C like `\n` for end-of-line. To count lines we might have the following,

```
\n  lineno++;
```

where **lineno**, like other C variables, is declared in the definitions section that we discuss later.

lex stores every character string that it recognizes in a character array called **yytext[]**. You can print or manipulate the contents of this array as you want. Sometimes your action may consist of two or more C statements and you must (or for style and clarity, you choose to) write it on several lines. To inform **lex** that the action is for one rule only, simply enclose the C code in braces. For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings (not their sum, here) and print out each one as soon as it is found, your **lex** code might be

```
+?[1-9]+      { digstrngcount++;
                printf("%d",digstrngcount);
                printf("%s", yytext);  }
```

This specification matches digit strings whether they are preceded by a plus sign or not, because the `?` indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings because that portion following the minus sign, `-`, will match the specification. The next section explains how to distinguish negative from positive integers.

Advanced lex Usage

lex provides a suite of features that lets you process input text riddled with quite complicated patterns. These include rules that decide what specification is relevant, when more than one seems so at first; functions that transform one matching pattern into another; and the use of definitions and subroutines. Before considering these features, you may want to affirm your understanding thus far by examining the following example drawing together several of the points already covered:

```
%%
-[0-9]+      printf("negative integer");
+?[0-9]+     printf("positive integer");
-0.[0-9]+   printf("negative fraction,
              no whole number part");
rail[ ]+road printf("railroad is one word");
crook       printf("Here's a crook");
function    subprogcount++;
G[a-zA-Z]*  { printf("may have a G word here: ", yytext);
              Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and -1. The use of the terminating + in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for **railroad** matches cases where one or more blanks intervene between the two syllables of the word. In the cases of **railroad** and **crook**, you may have simply printed a synonym rather than the messages stated. The rule recognizing a **function** simply increments a counter. The last rule illustrates several points:

- The braces specify an action sequence extending over several lines.

- Its action uses the **lex** array **yytext[]**, which stores the recognized character string.
- Its specification uses the ***** to indicate that zero or more letters may follow the **G**.

Some Special Features

Besides storing the recognized character string in **yytext[]**, **lex** automatically counts the number of characters in a match and stores it in the variable **yytext**. You may use this variable to refer to any specific character just placed in the array **yytext[]**. Remember that C numbers locations in an array starting with 0, so to print out the third digit (if there is one) in a just recognized integer, you might write

```
[1-9]+      {if (yytext > 2)
              printf("%c", yytext[2]); }
```

lex follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. *Prima facie*, any reserved word, for instance, could match two rules. In the lexical analyzer example developed later in the section on **lex** and **yacc**, the reserved word **end** could match the second rule as well as the seventh, the one for identifiers.

NOTE

lex follows the rule that where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed.

By placing the rule for **end** and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize **>** and **>=**. If the text has the string **>=** at one point, you might

Writing lex Programs

worry that the lexical analyzer would stop as soon as it recognized the `>` character to execute the rule for `>` rather than read the next character and execute the rule for `> =`.

NOTE

lex follows the rule that it matches the longest character string possible and executes the rule for that.

Here it would recognize the `> =` and act accordingly. As a further example, the rule would enable you to distinguish `+` from `++` in a program in C.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure you've in fact found it until you've read the additional characters. These cases reveal the importance of trailing context. The classic example here is the DO statement in FORTRAN. In the statement

```
DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index **k** until we read the first comma. Until then, we might have the assignment statement

```
DO50k = 1
```

(Remember that FORTRAN ignores all blanks.) The way to handle this is to use the forward-looking slash, `/` (not the backslash, `\`), which signifies that what follows is trailing context, something not to be stored in `yytext[]`, because it is not part of the token itself. So the rule to recognize the FORTRAN DO statement could be

```
DO/[ ]*[0-9][ ]*[a-zA-Z0-9]+=[a-zA-Z0-9]+, printf("found DO");
```

Different versions of FORTRAN have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length.

lex uses the **\$** as an operator to mark a special trailing context—the end of line. (It is therefore equivalent to **\n**.) An example would be a rule to ignore all blanks and tabs at the end of a line:

```
[ \t]+$ ;
```

On the other hand, if you want to match a pattern only when it starts a line, **lex** offers you the circumflex, **^**, as the operator. The formatter **nroff**, for example, demands that you never start a line with a blank, so you might want to check input to **nroff** with some such rule as:

```
^[ ] printf("error: remove leading blank");
```

Finally, some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three functions to handle these tasks—**input()**, **unput(c)**, and **output(c)**, respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use **input()**, thus:

```
\ " while (input() != "'");
```

Upon finding the first double quotation mark, the generated **a.out** will simply continue reading all subsequent characters so long as none is a quotation mark, and not again look for a match until it finds a second double quotation mark.

To handle special I/O needs, such as writing to several files, you may use standard I/O routines in C to rewrite the functions **input()**, **unput(c)**, and **output**. These and other programmer-defined functions should be placed in your subroutine section. Your new routines will then replace the standard ones. The standard **input()**, in fact, is equivalent to **getchar()**, and the standard **output(c)** is equivalent to **putchar(c)**.

Writing lex Programs

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. These include **yymore()**, **yyles(n)**, and **REJECT**. Recall that the text matching a given specification is stored in the array **yytext[]**. In general, once the action is performed for the specification, the characters in **yytext[]** are overwritten with succeeding characters in the input stream to form the next match. The function **yymore()**, by contrast, ensures that the succeeding characters recognized are appended to those already in **yytext[]**. This lets you do one thing and then another, when one string of characters is significant and a longer one including the first is significant as well. Consider a character string bound by **B**s and interspersed with one at an arbitrary location.

B...B...B

In a simple code deciphering situation, you may want to count the number of characters between the first and second **B**'s and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.) The code to do this is

```
B[ ^B]*      { if (flag = 0)
                save = yyleng;
                flag = 1;
                yymore();
            else  {
                importantno = save + yyleng;
                flag = 0; }
            }
```

where **flag**, **save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section. The **flag** distinguishes the character sequence terminating just before the second **B** from that terminating just before the third.

The function **yyles(n)** lets you reset the end point of the string to be considered to the *n*th character in the original **yytext[]**. Suppose you are again in the code deciphering business and the gimmick here is to work with only half the characters in a sequence ending with a certain one, say upper- or lowercase **Z**. The code you want might be:

```
[a-yA-Y]+[Zz] { yyles(yyleng/2);
                ... process first half of string... }
```

Finally, the function REJECT lets you more easily process strings of characters even when they overlap or contain one another as parts. REJECT does this by immediately jumping to the next rule and its specification without changing the contents of `yytext[]`. If you want to count the number of occurrences both of the regular expression **snapdragon** and of its subexpression **dragon** in an input text, the following will do:

```
snapdragon      {countflowers++; REJECT;}
dragon          countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions **comedian** and **diana**, even where the input text has sequences such as **comedi-ana..**:

```
comedian        {comiccount++; REJECT;}
diana          princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter. In all cases, the counters and other necessary variables are declared in the definitions section commencing the **lex** specification.

Definitions

The **lex** definitions section may contain any of several classes of items. The most critical are external definitions, **#include** statements, and abbreviations. Recall that for legal **lex** source this section is optional, but in most cases some of these items are necessary. External definitions have the form and function that they do in C. They declare that variables globally defined elsewhere (perhaps in another source file) will be accessed in your **lex**-generated **a.out**. Consider a declaration from an example to be developed later.

```
extern int tokval;
```

When you store an integer value in a variable declared in this way, it will be accessible in the routine, say a parser, that calls it. If, on the

Writing lex Programs

other hand, you want to define a local variable for use within the action sequence of one rule (as you might for the index variable for a loop), you can declare the variable at the start of the action itself right after the left brace, { .

The purpose of the **#include** statement is the same as in C: to include files of importance for your program. Some variable declarations and **lex** definitions might be needed in more than one **lex** source file. It is then advantageous to place them all in one file to be included in every file that needs them. One example occurs in using **lex** with **yacc**, which generates parsers that call a lexical analyzer. In this context, you should include the file **y.tab.h**, which may contain **#defines** for token names. Like the declarations, **#include** statements should come between `%{` and `%}`, thus:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, after the `%}` that ends your **#include**'s and declarations, you place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you later use abbreviations in your rules, be sure to enclose them within braces.

NOTE

The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them.

As an example, reconsider the **lex** source reviewed at the beginning of this section on advanced **lex** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times:


```

D          [0-9]
L          [a-zA-Z]
B          [ ]
%%
-{D}+     printf("negative integer");
+?{D}+    printf("positive integer");
-0.{D}+   printf("negative fraction");
G{L}*     printf("may have a G word here");
rail{B}+road printf("railroad is one word");
crook     printf("criminal");
  \\. /{B}+ printf(".\`"");
  :
  :
```

The last rule, newly added to the example and somewhat more complex than the others, is used in the WRITER'S WORKBENCH Software, an AT&T software product for promoting good writing. (See the *UNIX System WRITER'S WORKBENCH Software Release 3.0 User's Guide* for information on this product.) The rule ensures that a period always precedes a quotation mark at the end of a sentence. It would change example". to example."

Subroutines

You may want to use subroutines in **lex** for much the same reason that you do so in other programming languages. Action code that is to be used for several rules can be written once and called when needed. As with definitions, this can simplify the writing and reading of programs. The function **put_in_tabl()**, to be discussed in the next section on **lex** and **yacc**, is a good candidate for a subroutine.

Another reason to place a routine in this section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between `/*` and `*/` :

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX system program tool **yacc**. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in detail in Chapter 6 of this guide.) **lex** often forms a fruitful union with **yacc** in the compiler development context. Whether or not you plan to use **lex** with **yacc**, be sure to read this section because it covers information of interest to all **lex** programmers.

The lexical analyzer that **lex** generates (not the file that stores it) takes the name **yylex()**. This name is convenient because **yacc** calls its lexical analyzer by this very name. To use **lex** to create the lexical analyzer for the parser of a compiler, you want to end each **lex** action with the statement **return token**, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, whose file is called **y.tab.c** by **yacc**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operand, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant, say, is 9 or 888, whether the operand is + or * (multiply), and whether the relational operator is = or >.

Consider the following portion of **lex** source for a lexical analyzer for some programming language perhaps slightly reminiscent of Ada:

Writing lex Programs

```

begin                return(BEGIN);
end                  return(END);
while                return(WHILE);
if                   return(IF);
package              return(PACKAGE);
reverse              return(REVERSE);
loop                 return(LOOP);
[a-zA-Z][a-zA-Z0-9]* { tokval = put_in_tabl();
                       return(IDENTIFIER); }
[0-9]+               { tokval = put_in_tabl();
                       return(INTEGER); }
\+                   { tokval = PLUS;
                       return(ARITHOP); }
\-                   { tokval = MINUS;
                       return(ARITHOP); }
>                    { tokval = GREATER;
                       return(RELOP); }
>=                   { tokval = GREATEREQ;
                       return(RELOP); }

```

Despite appearances, the tokens returned, and the values assigned to **tokval**, are indeed integers. Good programming style dictates to use informative terms such as **BEGIN**, **END**, **WHILE**, and so forth to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using **#define** statements in your parser calling routine in C. For example,

```

#define BEGIN 1
#define END 2
.
#define PLUS 7
.

```

If the need arises to change the integer for some token type, you then change the **#define** statement in the parser rather than hunt through the entire program, changing every occurrence of the particular integer. In using **yacc** to generate your parser, it is helpful to

insert the statement

```
#include y.tab.h
```

into the definitions section of your **lex** source. The file **y.tab.h** provides **#define** statements that associate token names such as **BEGIN**, **END**, and so on with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable **tokval**. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable **yylval** for the same purpose.

Note that the example shows two ways to assign a value to **tokval**. First, a function **put_in_tabl()** places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, **put_in_tabl()** assigns a type value to **tokval** so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function **put_in_tabl()** would be a routine that the compiler writer might place in the sub-routines section discussed later. Second, in the last few actions of the example, **tokval** is assigned a specific integer indicating which operand or relational operator the analyzer recognized. If the variable **PLUS**, for instance, is associated with the integer 7 by means of the **#define** statement above, then when a **+** sign is recognized, the action assigns to **tokval** the value 7, which indicates the **+**. The analyzer indicates the general class of operator by the value it returns to the parser (in the example, the integer signified by **ARITHOP** or **RELOP**).

This page is intentionally left blank

Running `lex` under the UNIX System

As you review the following few steps, you might recall Figure 5-1 at the start of the chapter. To produce the lexical analyzer in C, run

```
lex lex.l
```

where `lex.l` is the file containing your `lex` specification. The name `lex.l` is conventionally the favorite, but you may use whatever name you want. The output file that `lex` produces is automatically called `lex.yy.c`; this is the lexical analyzer program that you created with `lex`. You then compile and link this as you would any C program, making sure that you invoke the `lex` library with the `-ll` option:

```
cc lex.yy.c -ll
```

The `lex` library provides a default `main()` program that calls the lexical analyzer under the name `yylex()`, so you need not supply your own `main()`.

If you have the `lex` specification spread across several files, you can run `lex` with each of them individually, but be sure to rename or move each `lex.yy.c` file (with `mv`) before you run `lex` on the next one. Otherwise, each will overwrite the previous one. Once you have all the generated `.c` files, you can compile all of them, of course, in one command line.

With the executable `a.out` produced, you are ready to analyze any desired input text. Suppose that the text is stored under the filename `textin` (this name is also arbitrary). The lexical analyzer `a.out` by default takes input from your terminal. To have it take the file `textin` as input, simply use redirection, thus:

```
a.out < textin
```

By default, output will appear on your terminal, but you can redirect this as well:

```
a.out < textin > textout
```

Running `lex` under the UNIX System

In running `lex` with `yacc`, either may be run first.

```
yacc -d grammar.y
lex lex.l
```

spawns a parser in the file `y.tab.c`. (The `-d` option creates the file `y.tab.h`, which contains the `#define` statements that associate the `yacc` assigned integer token values with the user-defined token names.) To compile and link the output files produced, run

```
cc lex.yy.c y.tab.c -ly -ll
```

Note that the `yacc` library is loaded (with the `-ly` option) before the `lex` library (with the `-ll` option) to ensure that the `main()` program supplied will call the `yacc` parser.

There are several options available with the `lex` command. If you use one or more of them, place them between the command name `lex` and the filename argument. If you care to see the C program, `lex.yy.c`, that `lex` generates on your terminal (the default output device), use the `-t` option.

```
lex -t lex.l
```

The `-v` option prints out for you a small set of statistics describing the so-called finite automata that `lex` produces with the C program `lex.yy.c`. (For a detailed account of finite automata and their importance for `lex`, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

`lex` uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 500. If your `lex` source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing another entry in the definitions section of your `lex` source, as follows:

```
%n 700
```

This entry tells `lex` to make the table large enough to handle as many as 700 states. (The `-v` option will indicate how large a number you

should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is **a**, thus:

```
%a 2800
```

Finally, check the *System V Reference Manual* page on **lex** for a list of all the options available with the **lex** command. In addition, review the paper by Lesk (the originator of **lex**) and Schmidt, "Lex—A Lexical Analyzer Generator," in volume 5 of the *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, 1986. It is somewhat dated, but offers several interesting examples.

This tutorial has introduced you to **lex** programming. As with any programming language, the way to master it is to write programs and then write some more.



Running lex under the UNIX System

This page is intentionally left blank

Chapter 6: yacc

	Page
Introduction.....	6- 1
Basic Specifications.....	6- 5
Actions	6- 8
Lexical Analysis	6-11
Parser Operation	6-15
Ambiguity and Conflicts.....	6-21
Precedence.....	6-27
Error Handling.....	6-31
The yacc Environment.....	6-35
Hints for Preparing Specifications.....	6-37
Input Style.....	6-37
Left Recursion.....	6-38
Lexical Tie-Ins	6-39
Reserved Words	6-41
Advanced Topics	6-43
Simulating error and accept in Actions	6-43
Accessing Values in Enclosing Rules.....	6-43
Support for Arbitrary Value Types.....	6-45
yacc Input Syntax.....	6-47
Examples.....	6-51
1. A Simple Example	6-51
2. An Advanced Example.....	6-55

Table of Contents

This page is intentionally left blank

Introduction

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes:

- a set of rules to describe the elements of the input
- code to be invoked when a rule is recognized
- either a definition or declaration of a low-level routine to examine the input

yacc then turns the specification into a C language function that examines the input stream. This function, called a parser, works by calling the low-level input scanner. The low-level input scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens. Tokens are compared to the input construct rules, called grammar rules. When one of the rules is recognized, the user code supplied for this rule, (an action) is invoked. Actions are fragments of C language code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

where **date**, **month_name**, **day**, and **year** represent constructs of interest; presumably, **month_name**, **day**, and **year** are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in evaluating the input. With proper definitions, the input

```
July 4, 1776
```

might be matched by the rule.

Introduction

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as non-terminal symbols. To avoid confusion, we will refer to terminal symbols as tokens.

There is considerable leeway in deciding whether to recognize constructs using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
...
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space, and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a **month_name** is seen. In this case, **month_name** is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7/4/1776
```

as a synonym for

July 4, 1776

on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. With a left-to-right scan input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases often can be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructs that are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The remainder of this chapter describes the following subjects:

- basic process of preparing a **yacc** specification
- parser operation
- handling ambiguities
- handling operator precedences in arithmetic expressions

Introduction

- error detection and recovery
- the operating environment and special features of the parsers **yacc** produces
- suggestions to improve the style and efficiency of the specifications
- advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

Basic Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to keep it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Thus, every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs, %% (the percent sign is generally used in **yacc** specifications as an escape character).

A full specification file looks like:

```
declarations
%%
rules
%%
subroutines
```

when all sections are used. The *declarations* and *subroutines* sections are optional. The smallest legal **yacc** specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A : BODY ;
```

where **A** represents a nonterminal symbol, and **BODY** represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Basic Specifications

Names may be of any length and may be made up of letters, dots, underscores, and digits although a digit may not be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes, '. As in the C language, the backslash, \, is an escape character within literals, and all the C language escapes are recognized. Thus:

```

'\n'  newline
'\r'  return
'\''  single quote ( ' )
'\'\'  backslash ( \ )
'\t'  tab
'\b'  backspace
'\f'  form feed
'\xxx' xxx in octal notation

```

are understood by **yacc**. For a number of technical reasons, the NULL character (\0 or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar, |, can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar. Thus the grammar rules

```

A   : B C D ;
A   : E F ;
A   : G ;

```

can be given to **yacc** as

```

A   : B C D
    | E F
    | G
    ;

```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by

```
epsilon : ;
```

The blank space following the colon is understood by **yacc** to be a nonterminal symbol named **epsilon**.

Names representing tokens must be declared. This is most simply done by writing

```
%token name1 name2 ...
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the start symbol has particular importance. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword.

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to but not including the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end of file or end of record.

Actions

With each grammar rule, the user may associate actions to be performed when the rule is recognized. Actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and as such can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in curly braces, {, and }. For example:

```
A   :  '( ' B ' )'
      {
        hello( 1, "abc" );
      }
```

and

```
XXX :  YYY ZZZ
      {
        (void) printf("a message\n");
        flag = 25;
      }
```

are grammar rules with actions.

The dollar sign symbol, \$, is used to facilitate communication between the actions and the parser. The pseudo-variable \$\$ represents the value returned by the complete action. For example, the action

```
{ $$ = 1; }
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables \$1, \$2, ... \$n. These refer to the values returned by components 1 through n of the right side of a rule, with the components being numbered from left to

right. If the rule is

```
A : B C D ;
```

then **\$2** has the value returned by **C**, and **\$3** the value returned by **D**.

The rule

```
expr : '(' expr ')' ;
```

provides a common example. One would expect the value returned by this rule to be the value of the *expr* within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by

```
expr : '(' expr ')'
      {
        $$ = $2 ;
      }
```

By default, the value of a rule is the value of the first element in it (**\$1**). Thus, grammar rules of the form

```
A : B ;
```

frequently need not have an explicit action. In previous examples, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This action is assumed to return a value accessible through the usual **\$** mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule below the effect is to set **x** to 1 and **y** to the value returned by **C**.

Basic Specifications

```

A : B
    {
        $$ = 1;
    }
    C
    {
        x = $2;
        y = $3;
    }
;

```

Actions that do not terminate a rule are handled by **yacc** by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered by recognizing this added rule. **yacc** treats the above example as if it had been written

```

$ACT : /* empty */
    {
        $$ = 1;
    }
;

A : B $ACT C
    {
        x = $2;
        y = $3;
    }
;

```

where **\$ACT** is an empty action.

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and

maintain the tree structure desired. For example, suppose there is a C function node written so that the call

```
node( L, n1, n2 )
```

creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly created node. Then a parse tree can be built by supplying actions such as

```
expr   :   expr '+' expr
        {
            $$ = node( '+', $1, $3 );
        }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks `%{` and `%}`. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example:

```
%{   int variable = 0;   %}
```

could be placed in the declarations section making **variable** accessible to all of the actions. Users should avoid names beginning with **yy** because the **yacc** parser uses only such names. In the examples shown thus far all the values are integers. A discussion of values of other types is found in the section "Advanced Topics."

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yylval**.

Basic Specifications

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **#define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name **DIGIT** has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like

```
int yylex()
{
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch (c)
    {
        ...
        case '0':
        case '1':
        ...
        case '9':
            yylval = c - '0';
            return (DIGIT);
        ...
    }
    ...
}
```

to return the appropriate token.

The intent is to return a token number of **DIGIT** and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the subroutines section of the specification file, the identifier **DIGIT** is defined as the token number associated with the token **DIGIT**.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

In the default situation, token numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257. If the **yacc** command is invoked with the **-d** option a file called **y.tab.h** is generated. **y.tab.h** contains **#define** statements for the tokens.

If the user prefers to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

For historical reasons, the end-marker must have token number 0 or negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the **lex** utility. Lexical analyzers produced by **lex** are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN), which do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

Basic Specifications

This page is intentionally left blank

Parser Operation

yacc turns the specification file into a C language procedure, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, though, is relatively simple and understanding its usage will make treatment of error recovery and ambiguities easier.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the look-ahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available—**shift**, **reduce**, **accept**, and **error**. A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look-ahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action

```
IF shift 34
```

which says, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

Parser Operation

The **reduce** action keeps the stack from growing without bounds. **reduce** actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether or not to **reduce** (usually it is not necessary). In fact, the default action (represented by a dot) is often a **reduce** action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The action

```
.   reduce 18
```

refers to grammar rule 18, while the action

```
IF   shift 34
```

refers to state 34.

Suppose the rule

```
A   :   x y z   ;
```

is being reduced. The **reduce** action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case).

To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing x, y, and z and no longer serve any useful purpose.

After popping these states, a state is uncovered, which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a **goto** action. In particular, the look-ahead token is cleared by a shift but is not affected by a **goto**. In any case, the uncovered state contains an entry such as

A goto 20

causing state 20 to be pushed onto the stack and become the current state. In effect, the **reduce** action turns back the clock in the parse popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks. The uncovered state is in fact the current state.

The **reduce** action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a **shift** takes place, the external variable **yyival** is copied onto the value stack. After the return from the user code, the reduction is carried out. When the **goto** action is done, the external variable **yyval** is copied onto the value stack. The pseudo-variables **\$1**, **\$2**, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job. The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Parser Operation

Consider:

```
%token DING DONG DELL
%%
rhyme : sound place
      ;
sound : DING DONG
      ;
place : DELL
      ;
```

as a **yacc** specification. When **yacc** is invoked with the **-v** option, a file called **y.output** is produced with a human-readable description of the parser. The **y.output** file corresponding to the above grammar (with some statistics stripped off the end) follows.

```
state 0
    $accept : _rhyme $end

    DING shift 3
    . error

    rhyme goto 1
    sound goto 2

state 1
    $accept : rhyme_$end

    $end accept
    . error

state 2
    rhyme : sound_place

    DELL shift 5
    . error
```

(continued on next page)

```

        place goto 4

state 3
    sound : DING_DONG

        DONG shift 6
        . error

state 4
    rhyme : sound place_ (1)

        . reduce 1

state 5
    place : DELL_ (3)

        . reduce 3

state 6
    sound : DING DONG_ (2)

        . reduce 2

```

The actions for each state are specified and there is a description of the parsing rules being processed in each state. The `_` character is used to indicate what has been seen and what is yet to come in each rule. The following input

```
DING DONG DELL
```

can be used to track the operations of the parser. Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read and becomes the look-ahead token. The action in state 0 on DING is **shift 3**, state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains

Parser Operation

0, 3, and 6. In state 6 the parser reduces by

```
sound : DING DONG
```

which is rule 2. Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a **goto** on **sound**),

```
sound goto 2
```

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, DELL, must be read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The **goto** in state 2 on **place** (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a **goto** on **rhyme** causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained indicated by **\$end** in the **y.output** file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

$$\text{expr} : \text{expr} \text{ '-' } \text{expr}$$

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

$$\text{expr} - \text{expr} - \text{expr}$$

the rule allows this input to be structured as either

$$(\text{expr} - \text{expr}) - \text{expr}$$

or as

$$\text{expr} - (\text{expr} - \text{expr})$$

(The first is called left association, the second right association.)

yacc detects such ambiguities when it is attempting to build the parser. Given the input

$$\text{expr} - \text{expr} - \text{expr}$$

consider the problem that confronts the parser. When the parser has read the second *expr*, the input seen

$$\text{expr} - \text{expr}$$

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input

$$- \text{expr}$$

and again reduce. The effect of this is to take the left associative interpretation.

Ambiguity and Conflicts

Alternatively, if the parser sees

$expr - expr$

it could defer the immediate application of the rule and continue reading the input until

$expr - expr - expr$

is seen. It could then apply the rule to the rightmost three symbols reducing them to *expr*, which results in

$expr - expr$

being left. Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read

$expr - expr$

the parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a **shift-reduce** conflict. It may also happen that the parser has a choice of two legal reductions. This is called a **reduce-reduce** conflict. Note that there are never any **shift-shift** conflicts.

When there are **shift-reduce** or **reduce-reduce** conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a disambiguating rule.

yacc invokes two default disambiguating rules:

1. In a **shift-reduce** conflict, the default is to do the shift.
2. In a **reduce-reduce** conflict, the default is to reduce by the earlier grammar rule (in the **yacc** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but **reduce-reduce** conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of **shift-reduce** and **reduce-reduce** conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```

stat      :   IF '(' cond ')' stat
           |   IF '(' cond ')' stat ELSE stat
           ;

```

which is a fragment from a programming language involving an **if-then-else** statement. In these rules, **IF** and **ELSE** are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple **if** rule and the second the **if-else** rule.

These two rules form an ambiguous construction because input of the form

```

IF ( C1 ) IF ( C2 ) S1 ELSE S2

```

can be structured according to these rules in two ways

Ambiguity and Conflicts

```

IF ( C1 )
{
    IF ( C2 )
        S1
}
ELSE
    S2

```

or

```

IF ( C1 )
{
    IF ( C2 )
        S1
    ELSE
        S2
}

```

where the second interpretation is the one given in most programming languages having this construct; each **ELSE** is associated with the last preceding un-**ELSE**'d **IF**. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the **ELSE**. It can immediately reduce by the simple **if** rule to get

```
IF ( C1 ) stat
```

and then read the remaining input

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the **if-else** rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right-hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple **if** rule. This leads to the second of the above groupings of the input which is usually desired.

Once again, the parser can do two valid things – there is a **shift-reduce** conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This **shift-reduce** conflict arises only when there is a particular current input symbol, ELSE, and particular inputs, such as

```
IF ( C1 ) IF ( C2 ) S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**-v**) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE

state 23

stat : IF ( cond ) stat_      (18)
stat : IF ( cond ) stat_ELSE stat

ELSE  shift 45
      .      reduce 18
```

Ambiguity and Conflicts

where the first line describes the conflict — giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules, which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat .
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

because the ELSE will have been shifted in this state. In state 23, the alternative action (describing a dot, .), is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not ELSE, the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammar rule 18.

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the **y.output** file, the rule numbers are printed in parentheses after those rules, which can be reduced. In most states, there is a reduce action possible in the state and this is the default command. The user who encounters unexpected **shift-reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The

Precedence

keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | NAME
     ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows

```
a = ( b = ( (c*d)-e) - (f*g) ) )
```

in order to perform the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus, -.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```

%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr %prec '*'
     | NAME
     ;

```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a **reduce-reduce** conflict or there is a **shift-reduce** conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a **shift-reduce** conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action—**shift** or **reduce**—associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; nonassociating implies **error**.

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in a cookbook fashion until some experience has been gained. The **y.output** file is very useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and/or, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides the token name **error**. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings

Error Handling

of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
stat : error ';' ;
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any cleanup action associated with it performed.

Another form of **error** rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example

```
input : error '\n'
      {
          (void) printf( "Reenter last line: " );
      }
      input
      {
          $$ = $4;
      }
      ;
```

is one way to do this. There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can

force the parser to believe that error recovery has been accomplished.

The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input : error '\n'
      {
        yyerrok;
        (void) printf( "Reenter last line: " );
      }
      input
    {
      $$ = $4;
    }
    ;
```

which is somewhat better.

As previously mentioned, the token seen immediately after the **error** symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after **error** were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by **yylex** is presumably the first token in a legal statement. The old illegal token must be discarded and the **error** state reset.

Error Handling

A rule similar to

```
stat : error
      {
        resynch();
        yyerrok ;
        yyclearin;
      }
      ;
```

could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C language subroutines, called **y.tab.c**. The function produced by **yacc** is called **yyparse()**; (an integer valued function). When it is called, it in turn repeatedly calls **yylex()**, the lexical analyzer supplied by the user (see "Lexical Analysis"), to obtain input tokens. If an error is detected, **yyparse()** returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, **yyparse()** returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a routine called **main()** must be defined that eventually calls **yyparse()**. In addition, a routine called **yyerror()** is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of **main()** and **yyerror()**. The library is accessed by a **-ly** argument to the **cc(1)** command or to the loader. The source codes

```
main()
{
    return (yyparse());
}

and

# include <stdio.h>
yyerror(s)
    char *s;
{
    (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs.

The yacc Environment

The argument to **yyerror()** is a string containing an error message, usually the string **syntax error**. The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the **main()** routine is probably supplied by the user (to read arguments, etc.), the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are. It is possible to set this variable by using **sdb**.

Hints for Preparing Specifications

This part contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This is useful in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

Example 1 is written following this style, as are the examples in this section (where space permits). The user must decide about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

Hints for Preparing Specifications

Left Recursion

The algorithm used by the **yacc** parser encourages so called left recursive grammar rules. Rules of the form

```
name : name rest_of_rule ;
```

match this algorithm. These rules such as

```
list : item
      | list ',' item
      ;
```

and

```
seq : item
      | seq item
      ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq : item
      | item seq
      ;
```

the parser is a bit bigger; and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq : /* empty */
      | seq item
      ;
```

using an empty rule. Once again, the first rule would always be

reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

Hints for Preparing Specifications

```
%{
    int dflag;
%}
... other declarations ...

%%

prog : decls stats
    ;

decls : /* empty */
    {
        dflag = 1;
    }
    | decls declaration
    ;

stats : /* empty */
    {
        dflag = 0;
    }
    | stats statement
    ;

... other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. The flag **dflag** is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back-door approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit you to use words like **if**, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it this instance of **if** is a keyword and that instance is a variable. The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be reserved, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Hints for Preparing Specifications

This page is intentionally left blank

Advanced Topics

This part discusses a number of advanced features of **yacc**.

Simulating error and accept in Actions

The parsing actions of **error** and **accept** can be simulated in an action by use of macros **YYACCEPT** and **YYERROR**. The **YYACCEPT** macro causes **yyparse()** to return the value 0; **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error; **yyerror()** is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit.

Advanced Topics

```
sent : adj noun verb adj noun
      {
        look at the sentence ...
      }
      ;
adj   : THE
      {
        $$ = THE;
      }
      | YOUNG
      {
        $$ = YOUNG;
      }
      ...
      ;
noun  : DOG
      {
        $$ = DOG;
      }
      | CRONE
      {
        if( $0 == YOUNG )
        {
          (void) printf( "what?\n" );
        }
        $$ = CRONE;
      }
      ;
      ...
```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked. **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names with each token and nonterminal symbol having a value. When the value is referenced through a **\$\$** or **\$n** construction, **yacc** will automatically insert the appropriate union name so that no unwanted conversions take place. In addition, type checking commands such as **lint** are far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other subroutines, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes

```
%union
{
    body of union ...
}
```

in the declaration section. This declares the **yacc** value stack and the external variables **yyval** and **yyval** to have type equal to this union. If **yacc** was invoked with the **-d** option, the union declaration is copied onto the **y.tab.h** file as **YYSTYPE**.

Once **YYSTYPE** is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

Advanced Topics

<name>

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name **optype**. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**.

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as **\$0**) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between **<** and **>** immediately after the first **\$**. The example

```
rule : aaa
      {
        $<intval>$ = 3;
      }
      bbb
    {
      fun( $<intval>2, $<other>0 );
    }
    ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of **\$n** or **\$\$** to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold **ints**.

yacc Input Syntax

This section has a description of the **yacc** input syntax as a **yacc** specification. Context dependencies, etc. are not considered. Ironically, although **yacc** accepts an LALR(1) grammar, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token **C_IDENTIFIER**. Otherwise, it returns **IDENTIFIER**. Literals (quoted strings) are also returned as **IDENTIFIERS** but never as part of **C_IDENTIFIERS**.

Advanced Topics

```

/* grammar for the input to yacc */

/* basic entries */
%token IDENTIFIER /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier (but not literal) followed by a : */
%token NUMBER /* [0-9]+ */

/* reserved words: %type=>TYPE %left=>LEFT,etc. */
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK /* the %% mark */
%token LCURL /* the %{ mark */
%token RCURL /* the %} mark */
/* ASCII character literals stand for themselves */

%token spec

%%

spec : defs MARK rules tail
;
tail : MARK
{
    In this action, eat up the rest of the file
}
| /* empty: the second MARK is optional */
;

defs : /* empty */
| defs def
;

def : START IDENTIFIER
| UNION
{
    Copy union definition to output
}
| LCURL
{
    Copy C code to output file
}

```

(continued on next page)

```

        RCURL
    |   rword tag nlist
    ;

rword :  TOKEN
    |   LEFT
    |   RIGHT
    |   NONASSOC
    |   TYPE
    ;

tag   :  /* empty: union tag is optional */
    |   '<' IDENTIFIER '>'
    ;

nlist :  nmno
    |   nlist nmno
    |   nlist ',' nmno
    ;

nmno  :  IDENTIFIER      /* Note: literal illegal with % type */
    |   IDENTIFIER NUMBER /* Note: illegal with % type */
    ;

/* rule section */

rules :  C IDENTIFIER rbody prec
    |   rules rule
    ;

rule  :  C IDENTIFIER rbody prec
    |   '|' rbody prec
    ;

rbody :  /* empty */
    |   rbody IDENTIFIER
    |   rbody act
    ;

act   :  '{'
    |   {
            Copy action translate $$ etc.
        }
    |   '}'
    ;

```

(continued on next page)

Advanced Topics

```
prec : /* empty */  
      | PREC IDENTIFIER  
      | PREC IDENTIFIER act  
      | prec ';' ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled **a** through **z** and accepts arithmetic expressions made up of the operators

`+`, `-`, `*`, `/`, `%` (mod operator), `&` (bitwise and),
`|` (bitwise or), and assignments.

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

Examples

```

%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* supplies precedence for unary minus */

%%      /* beginning of rules section */

list   : /* empty */
        | list stat '\n'
        | list error '\n'
        {
          yyerrok;
        }
        ;

stat   : expr
        {
          (void) printf( "%d\n", $1 );
        }
        | LETTER '=' expr
        {
          regs[$1] = $3;
        }
        ;

expr   : '(' expr ')'
        {
          $$ = $2;
        }

```

(continued on next page)


```
    }
    | expr '+' expr
    {
        $$ = $1 + $3;
    }
    | expr '-' expr
    {
        $$ = $1 - $3;
    }
    | expr '*' expr
    {
        $$ = $1 * $3;
    }
    | expr '/' expr
    {
        $$ = $1 / $3;
    }
    | expr '%' expr
    {
        $$ = $1 % $3;
    }
    | expr '&' expr
    {
        $$ = $1 & $3;
    }
    | expr '|' expr
    {
        $$ = $1 | $3;
    }
    | '-' expr %prec UMINUS
    {
        $$ = -$2;
    }
    | LETTER
    {
        $$ = reg[$1];
    }
    | number
    ;

number : DIGIT
```

(continued on next page)

Examples

```

    {
        $$ = $1; base = ($1= =0) ? 8 ; 10;
    }
    | number DIGIT
    {
        $$ = base * $1 + $2;
    }
    ;

%%          /* beginning of subroutines section */

int yylex( ) /* lexical analysis routine */
{
    /* return LETTER for lowercase letter, */
    /* yyval = 0 through 25 */
    /* returns DIGIT for digit, yyval = 0 through 9 */
    /* all other characters are returned immediately */
    int c;
        /*skip blanks*/
    while ((c = getchar()) = = ' ')
        ;

        /* c is now nonblank */

    if (islower(c))
    {
        yyval = c - 'a';
        return (LETTER);
    }
    if (isdigit(c))
    {
        yyval = c - '0';
        return (DIGIT);
    }
    return (c);
}
}

```

2. An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator example in Example 1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations $+$, $-$, $*$, $/$, unary $-$ **a** through **z**. Moreover, it also understands intervals written

$$(X, Y)$$

where **X** is less than or equal to **Y**. There are 26 interval valued variables **A** through **Z** that may also be used. The usage is similar to that in Example 1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, **INTERVAL**, by using **typedef**. **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of **YYERROR** to handle error conditions—division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 **shift-reduce** and 26 **reduce-reduce**. The problem can be seen by looking at the two input lines.

Examples

$$2.5 + (3.5 - 4.)$$

and

$$2.5 + (3.5, 4)$$

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator – one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C language library routine `atof()` is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence error recovery.

```

%{
#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
    double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();
double atof();
double dreg[26];
INTERVAL vreg[26];

%}

%start line

%union
{
    int ival;
    double dval;
    INTERVAL vval;
}

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST /* floating point constant */

%type <dval> dexp /* expression */

%type <vval> vexp /* interval expression */

/* precedence information about the operators */
%left '+' '-'
%left '*' '/'
%left UMINUS /* precedence for unary minus */

%% /* beginning of rules section */

lines : /* empty */
      | lines line
      ;

line : dexp '\n'

```

(continued on next page)

Examples

```

    {
        (void) printf("%15.8f\n", $1);
    }
    | vexp '\n'
    {
        (void) printf("%15.8f, %15.8f)\n", $1.lo, $1.hi);
    }
    | DREG '=' dexp '\n'
    {
        dreg[$1] = $3;
    }
    | VREG '=' vexp '\n'
    {
        vreg[$1] = $3;
    }
    | error '\n'
    {
        yyerrok;
    }
    ;
dexp
: CONST
| DREG
{
    $$ = dreg[$1];
}
| dexp '+' dexp
{
    $$ = $1 + $3;
}
| dexp '-' dexp
{
    $$ = $1 - $3;
}
| dexp '*' dexp
{
    $$ = $1 * $3;
}
| dexp '/' dexp
{

```

(continued on next page)

```

        $$ = $1 / $3;
    }
    | '-' dexp %prec UMINUS
    {
        $$ = -$2;
    }
    | '(' dexp ')'
    {
        $$ = $2;
    }
    ;

vexp : dexp
    {
        $$ .hi = $$ .lo = $1;
    }
    | '(' dexp ',' dexp ')'
    {
        $$ .lo = $2;
        $$ .hi = $4;
        if( $$ .lo > $$ .hi )
        {
            (void) printf("interval out of order \n");
            YYERROR;
        }
    }
    | VREG
    {
        $$ = vreg[$1];
    }
    | vexp '+' vexp
    {
        $$ .hi = $1 .hi + $3 .hi;
        $$ .lo = $1 .lo + $3 .lo;
    }
    | dexp '+' vexp
    {
        $$ .hi = $1 + $3 .hi;
        $$ .lo = $1 + $3 .lo;
    }
    | vexp '-' vexp

```

(continued on next page)

Examples

```

{
    $$hi = $1hi - $3lo;
    $$lo = $1lo - $3hi;
}
| dexp '-' vdep
{
    $$hi = $1 - $3lo;
    $$lo = $1 - $3hi
}
| vexp '*' vexp
{
    $$ = vmul( $1lo,$hi,$3 )
}
| dexp '*' vexp
{
    $$ = vmul( $1, $1, $3 )
}
| vexp '/' vexp
{
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1lo, $1hi, $3 )
}
| dexp '/' vexp
{
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1lo, $1hi, $3 )
}
| '-' vexp %prec UMINUS
{
    $$hi = -$2lo; $$lo = -$2hi
}
| '(' vexp ')'
{
    $$ = $2
}
}
;

%%                                /* beginning of subroutines section */

# define BSZ 50 /* buffer size for floating point number */

```

(continued on next page)


```
/* lexical analysis */

int yylex( )
{
    register int c;

    /* skip over blanks */
    while ((c = getchar()) == ' ')
        ;
    if (isupper(c))
    {
        yylval.ival = c - 'A';
        return (VREG);
    }

    if (islower(c))
    {
        yylval.ival = c - 'a',
        return( DREG );
    }

    /* gobble up digits, points, exponents */
    if (isdigit(c) || c == '.')
    {
        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for(; (cp - buf) < BSZ ; ++cp, c = getchar())
        {
            *cp = c;
            if (isdigit(c))
                continue;
            if (c == '.')
            {
                if (dot++ || exp)
                    return ('.'); /* will cause syntax error */
                continue;
            }
            if (c == 'e')
            {
                if (exp++)

```

(continued on next page)

Examples

```

        return ('e'); /* will cause syntax error */
        continue;
    }
    /* end of number */
    break;
}
*cp = ' ';
if (cp - buf >= BSZ)
    (void) printf("constant too long - truncated\n");
else
    ungetc(c, stdin); /* push back last char read */
    yylval.dval = atof(buf);
    return (CONST);
}
return (c);
}
INTERVAL
hilo(a, b, c, d)
    double a, b, c, d;
{
    /* returns the smallest interval containing a, b, c, and d */

    /* used by *,/ routine */
    INTERVAL v;

    if (a > b)
    {
        v.hi = a;
        v.lo = b;
    }
    else
    {
        v.hi = b;
        v.lo = a;
    }
    if (c > d)
    {
        if (c > v.hi)
            v.hi = c;
        if (d < v.lo)
            v.lo = d;
    }
}

```

(continued on next page)

```
else
}
    if (d > v.hi)
        v.hi = d;
    if (c < v.lo)
        v.lo = c;
}
return (v);
}
INTERVAL
vmul(a, b, v)
double a, b;
INTERVAL v;
{
    return (hilo(a * v.hi, a * v.lo, b * v.hi, b * v.lo));
}
dcheck(v)
INTERVAL v;
{
    if (v.hi >= 0. && v.lo <= 0.)
    {
        (void) printf("divisor interval contains 0.\n");
        return (1);
    }
    return (0);
}
INTERVAL
vdiv(a, b, v)
double a, b;
INTERVAL v;
{
    return (hilo(a / v.hi, a / v.lo, b / v.hi, b / v.lo));
}
```

Examples

This page is intentionally left blank

Chapter 7: File and Record Locking

	Page
Introduction.....	7- 1
Terminology.....	7- 3
File Protection.....	7- 5
Opening a File for Record Locking.....	7- 5
Setting a File Lock.....	7- 7
Setting and Removing Record Locks.....	7-10
Getting Lock Information.....	7-14
Deadlock Handling.....	7-17
Selecting Advisory or Mandatory Locking.....	7-18
Caveat Emptor - Mandatory Locking.....	7-19
Record Locking and Future Releases of the UNIX System .	7-20

Table of Contents

This page is intentionally left blank

Introduction

Mandatory and advisory file and record locking both are available on current releases of the UNIX system. The intent of this capability is to provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi-user applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

The remainder of this chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl**(2) system call, the **lockf**(3) library function, and **fcntl**(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Introduction

This page is intentionally left blank

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record

A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes

Processes that work together in some well defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Terminology

Advisory Locking

A form of record locking that does not interact with the I/O subsystem (i.e. **creat(2)**, **open(2)**, **read(2)**, and **write(2)**). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat(2)**, **open(2)**, **read(2)**, and **write(2)** system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for UNIX system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit (see **chmod(1)**) of the data base accessing programs. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail(1)** command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. For our example we will open our file for both read and write access:

File Protection

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

int fd;          /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
    extern void exit(), perror();

    /* get data base file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
    .
    .
    .
}
```

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend upon how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl(2)** system call, the other using the */usr/group* standards compatible **lockf(3)** library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl(2)** system call is as follows:

File Protection

```

#include <fcntl.h>
#define MAX_TRY 10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
 * is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK; /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L; /* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up.
 */
while (fcntl(fd, F_SETLK, &lck) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            (void) sleep(2);
            continue;
        }
        (void) fprintf(stderr, "File busy try again later!\n");
        return;
    }
    perror("fcntl");
    exit(2);
}

.
.
.

```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because `MAX_TRY` has been exceeded

To perform the same task using the `lockf(3)` function, the code is as follows:

```

#include <unistd.h>
#define MAX_TRY 10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up. */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again. */
        if (++try < MAX_TRY) {
            sleep(2);
            continue;
        }
        (void) fprintf(stderr, "File busy try again later!\n");
        return;
    }
    perror("lockf");
    exit(2);
}
.
.
.

```

It should be noted that the `lockf(3)` example appears to be simpler, but the `fcntl(2)` example exhibits additional flexibility. Using the `fcntl(2)` method, it is possible to set the type and start of the lock request simply by setting a few structure variables. `lockf(3)` merely sets write (exclusive) locks; an additional system call (`lseek(2)`) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a

read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the `/usr/group lockf` function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```

struct record {
    .
    .
    .
    long prev;      /* index to previous record in the list */
    long next;     /* index to next record in the list */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there are read
 * locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *   Set a write lock on "this".
 *   Return index to "this" record.
 * If any write lock is not obtained:
 *   Restore read locks on "here" and "next".
 *   Remove all other locks.
 *   Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
    struct flock lck;
    lck.l_type = F_WRLCK;      /* setting a write lock */
    lck.l_whence = 0; /* offset l_start from beginning of file */

    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* promote lock on "here" to write lock */
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "this" failed;

```

(continued on next page)

```

        * demote lock on "here" to read lock.
        */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "next" failed;
        * demote lock on "here" to read lock,
        */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* and remove lock on "this".
        */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1); /* cannot set lock, try again or quit */
    }
    return (this);
}

```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the `F_SETLKW` command. If the `F_SETLK` command was used instead, the `fcntl` system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the `lockf` function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that there are
 * no locks on "here" and "next".
 * If locks are obtained:
 *   Set a lock on "this".
 *   Return index to "this" record.
 * If any lock is not obtained:
 *   Remove all other locks.
 *   Return a -1.
 */
#include <unistd.h>
long
set3lock (this, here, next)
long this, here, next;
{
    /* lock "here" */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* lock "this" */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "this" failed.
         * Clear lock on "here".
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
    }

    /* lock "next" */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "next" failed.
         * Clear lock on "here",
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        /* and remove lock on "this".
         */
    }
}
```

(continued on next page)

```
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1); /* cannot set lock, try again or quit */
    }
    return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (`F_UNLCK` or `F_ULOCK`). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by `lck`. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous examples and the `F_GETLK` command is used in the `fcntl` call. If the lock passed to `fcntl` would be blocked, the first blocking lock is returned to the process through the structure passed to `fcntl`. That is, the lock data passed to `fcntl` is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, `l_pid` and `l_sysid`, that are only used by `F_GETLK`. (For systems that do not support a distributed architecture the value in `l_sysid` should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the **F_GETLK** command would not be blocked by another process' lock, then the **l_type** field is changed to **F_UNLCK** and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

```

struct flock lck;

/* Find and print "write lock" blocked segments of this file. */
(void) printf("sysid pid type start length\n");
lck.l_whence = 0;
lck.l_start = 0L;
lck.l_len = 0L;
do {
    lck.l_type = F_WRLCK;
    (void) fcntl(fd, F_GETLK, &lck);
    if (lck.l_type != F_UNLCK) {
        (void) printf("%5d %5d %c %8d %8d\n",
            lck.l_sysid,
            lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R',
            lck.l_start,
            lck.l_len);
        /* if this lock goes to the end of the address
         * space, no need to look further, so break out.
         */
        if (lck.l_len == 0)
            break;
        /* otherwise, look for new lock after the one
         * just found.
         */
        lck.l_start += lck.l_len;
    }
} while (lck.l_type != F_UNLCK);

```

fcntl with the **F_GETLK** command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

File Protection

The **lockf** function with the **F_TEST** command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using **lockf** to test for a lock on a file follows:

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
            (void) printf("file is locked by another process\n");
            break;
        case EBADF:
            /* bad argument passed to lockf */
            perror("lockf");
            break;
        default:
            (void) printf("lockf: unknown error <#d>\n", errno);
            break;
    }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem

appears in the **lockf(3)** function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection it should set its locks using **F_GETLK** instead of **F_GETLKW**.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made and the state of the permissions on the file (see **chmod(2)**). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;

    .
    .
    .
    if (stat(filename, &buf) < 0) {
        perror("program");
        exit (2);
    }
    /* get currently set mode */
    mode = buf.st_mode;
    /* remove group execute permission from mode */
    mode &= ~(S_IXEXEC>>3);
    /* set 'set group id bit' in mode */
    mode |= S_ISGID;
    if (chmod(filename, mode) < 0) {
        perror("program");
        exit(2);
    }

    .
    .
    .
```


Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod(1)** command can also be easily used to set a file to have mandatory locking. This can be done with the command:

```
chmod +l filename
```

The **ls(1)** command was also changed to show this setting when you ask for the long listing format:

```
ls -l filename
```

causes the following to be printed:

```
-rw---l--- 1 abc other 1048576 Dec 3 11:44 filename
```

Caveat Emptor – Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of the UNIX System

Provisions have been made for file and record locking in a UNIX system environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the **sleep-when-blocked** features of **fcntl** or **lockf** and that the process maintain its own deadlock detection. If the process uses the **sleep-when-blocked** feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Chapter 8: Interprocess Communication

	Page
Introduction.....	8- 1
Messages	8- 3
Getting Message Queues.....	8- 8
Using msgget	8- 8
Example Program.....	8-13
Controlling Message Queues.....	8-17
Using msgctl	8-17
Example Program.....	8-18
Operations for Messages	8-24
Using msgop	8-24
Sending a Message	8-24
Receiving a Message.....	8-26
Example Program.....	8-27
msgsnd	8-29
msgrcv	8-31
Semaphores	8-39
Using Semaphores.....	8-42
Getting Semaphores.....	8-46
Using semget	8-46
Example Program.....	8-50
Controlling Semaphores.....	8-54
Using semctl	8-55
Example Program.....	8-57
Operations on Semaphores.....	8-68

Table of Contents

	Page
Using semop	8-68
Example Program	8-70
Shared Memory	8-77
Using Shared Memory	8-78
Getting Shared Memory Segments	8-82
Using shmget	8-83
Example Program	8-87
Controlling Shared Memory	8-93
Using shmctl	8-93
Example Program	8-94
Operations for Shared Memory	8-103
Using shmop	8-103
Attaching a Shared Memory Segment	9-103
Detaching Shared Memory Segments	9-104
Example Program	8-105
shmat	8-106
shmdt	8-106

Introduction

The UNIX system supports three types of Inter-Process Communication (IPC):

- messages
- semaphores
- shared memory

This chapter describes the system calls for each type of IPC.

Included in the chapter are several example programs that show the use of the IPC system calls.

Since there are many ways in the C Programming Language to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that the calls provide.



Introduction

This page is intentionally left blank

Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can perform two operations:

- sending
- receiving

Before a message can be sent or received by a process, a process must have the UNIX operating system generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget(2)** system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl(2)** system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl()** to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process which is attempting to send a message can wait until the process which is to receive the message is ready and vice versa. A process which specifies that execution is to be suspended is performing a "blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

Messages

- It is successful.
- It receives a signal.
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (-1) is returned to the process, and an external error number variable **errno** is set accordingly.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier created is called the message queue identifier (**msqid**); it is used to identify or reference the associated message queue and data structure.

The message queue is used to store (header) information about each message that is being sent or received. This information includes the following for each message:

- pointer to the next message on queue
- message type
- message text size
- message text address

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- pointer to first message on the queue

- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

NOTE

All include files discussed in this chapter are located in the `/usr/include` or `/usr/include/sys` directories.

The C Programming Language data structure definition for the message information contained in the message queue is as follows:

```
struct msg
{
    struct msg      *msg_next; /* ptr to next message on q */
    long            msg_type;  /* message type */
    short           msg_ts;    /* message text size */
    short           msg_spot;  /* message text map address */
};
```

It is located in the `/usr/include/sys/msg.h` header file.

Likewise, the structure definition for the associated data structure is as follows:

```

struct msgid_ds
{
    struct ipc_perm  msg_perm;    /* operation permission struct */
    struct msg      *msg_first;  /* ptr to first message on q */
    struct msg      *msg_last;   /* ptr to last message on q */
    ushort         msg_cbytes;   /* current # bytes on q */
    ushort         msg_qnum;     /* # of messages on q */
    ushort         msg_qbytes;   /* max # of bytes on q */
    ushort         msg_lspid;    /* pid of last msgsnd */
    ushort         msg_lrpid;    /* pid of last msgrcv */
    time_t         msg_stime;    /* last msgsnd time */
    time_t         msg_rtime;    /* last msgrcv time */
    time_t         msg_ctime;    /* last change time */
};

```

It is located in the **#include <sys/msg.h>** header file also. Note that the **msg_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 8-1.

The definition of the **ipc_perm** data structure is as follows:

```

struct ipc_perm
{
    ushort  uid;    /* owner's user id */
    ushort  gid;    /* owner's group id */
    ushort  cuid;   /* creator's user id */
    ushort  cgid;   /* creator's group id */
    ushort  mode;   /* access modes */
    ushort  seq;    /* slot usage sequence number */
    key_t   key;    /* key */
};

```

Figure 8-1: **ipc_perm** Data Structure

It is located in the **#include <sys/ipc.h>** header file; it is common for all IPC facilities.

The **msgget(2)** system call is used to perform two tasks when only the **IPC_CREAT** flag is set in the **msgflg** argument that it receives:

- to get a new **msqid** and create an associated message queue and data structure for it
- to return an existing **msqid** that already has an associated message queue and data structure

The task performed is determined by the value of the **key** argument passed to the **msgget()** system call. For the first task, if the **key** is not already in use for an existing **msqid**, a new **msqid** is returned with an associated message queue and data structure created for the **key**. This occurs provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (**IPC_PRIVATE = 0**); when specified, a new **msqid** is always returned with an associated message queue and data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, for security reasons the **KEY** field for the **msqid** is all zeros.

For the second task, if a **msqid** exists for the **key** specified, the value of the existing **msqid** is returned. If you do not desire to have an existing **msqid** returned, a control command (**IPC_EXCL**) can be specified (set) in the **msgflg** argument passed to the system call. The details of using this system call are discussed in the "Using **msgget**" section of this chapter.

When performing the first task, the process which calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see the "Controlling Message Queues" section in this chapter. The creator of the message queue also determines the initial operation permissions for it.

Messages

Once a uniquely identified message queue and data structure are created, message operations [**msgop**(2)] and message control [**msgctl**()] can be used.

Message operations, as mentioned previously, consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd**() and **msgrcv**(). Refer to the "Operations for Messages" section in this chapter for details of these system calls.

Message control is done by using the **msgctl**(2) system call. It permits you to control the message facility in the following ways:

- to determine the associated data structure status for a message queue identifier (**msqid**)
- to change operation permissions for a message queue
- to change the **size** (**msg_qbytes**) of the message queue for a particular **msqid**
- to remove a particular **msqid** from the UNIX operating system along with its associated message queue and data structure

Refer to the "Controlling Message Queues" section in this chapter for details of the **msgctl**() system call.

Getting Message Queues

This section gives a detailed description of using the **msgget**(2) system call along with an example program illustrating its use.

Using msgget

The synopsis found in the **msgget**(2) entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

All of these include files are located in the `/usr/include/sys` directory of the UNIX operating system.

The following line in the synopsis:

```
int msgget (key, msgflg)
```

informs you that `msgget()` is a function with two formal arguments that returns an integer type value, upon successful completion (`msqid`). The next two lines:

```
key_t key;
int msgflg;
```

declare the types of the formal arguments. `key_t` is declared by a `typedef` in the `types.h` header file to be an integer.

The integer returned from this function upon successful completion is the message queue identifier (`msqid`) that was discussed earlier.

Messages

As declared, the process calling the **msgget()** system call must supply two arguments to be passed to the formal **key** and **msgflg** arguments.

A new **msqid** with an associated message queue and data structure is provided if either

- **key** is equal to `IPC_PRIVATE`,
- or
- **key** is passed a unique hexadecimal integer, and **msgflg** ANDed with `IPC_CREAT` is `TRUE`.

The value passed to the **msgflg** argument must be an integer type octal value and it will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **msgflg** argument. They are collectively referred to as "operation permissions." Figure 8-2 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 8-2: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **msg.h** header file which can be used for the user (OWNER).

Control commands are predefined constants (represented by all uppercase letters). Figure 8-3 contains the names of the constants which apply to the **msgget()** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 8-3: Control Commands (Flags)

The value for **msgflg** is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (**|**) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
Read by User	=	0 0 4 0 0	0 000 000 100 000 000
msgflg	=	0 1 4 0 0	0 000 001 100 000 000

The **msgflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));  
  
msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **msgget(2)** page in the *System V Reference Manual*, success or failure of this system call depends upon the argument values for **key** and **msgflg** or system tunable parameters. The system call will attempt to return a new **msqid** if one of the following conditions is true:

- Key is equal to **IPC_PRIVATE** (0)
- Key does not already have a **msqid** associated with it, and (**msgflg & IPC_CREAT**) is "true" (not zero).

The **key** argument can be set to **IPC_PRIVATE** in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);
```

or

```
msqid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the **MSGMNI** system tunable parameter always causes a failure. The **MSGMNI** system tunable parameter determines the maximum number of unique message queues (**msqid**'s) in the UNIX operating system.

The second condition is satisfied if the value for **key** is not already associated with a **msqid** and the bitwise ANDing of **msgflg** and **IPC_CREAT** is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the **IPC_CREAT** flag is set (**msgflg | IPC_CREAT**). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:


```

msgflg           =    x 1 x x x   (x = immaterial)
& IPC_CREAT       =    0 1 0 0 0

result            =    0 1 0 0 0   (not zero)

```

Since the result is not zero, the flag is set or "true."

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **msgid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msgid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **msgid** is returned if the system call is successful.

Refer to the **msgget(2)** page in the *System V Reference Manual* for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 8-4) is a menu driven program which allows all possible combinations of using the **msgget(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **msgget(2)** entry in the *System V Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables

Messages

declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument
- **msqid**—used for returning the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (**flags**) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-51).

The system call is made next, and the result is stored at the address of the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 55). If **msqid** equals -1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the **msgget(2)** system call follows. It is suggested that the source program file be named **msgget.c** and that the executable file be named **msgget**.

Figure 8-4: `msgget()` System Call Example

```
1  /*This is a program to illustrate
2  **the message get, msgget(),
3  **system call capabilities.*/

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/msg.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key;           /*declare as long integer*/
13     int opperm, flags;
14     int msqid, opperm_flags;
15     /*Enter the desired key*/
16     printf("Enter the desired key in hex = ");
17     scanf("%x", &key);

18     /*Enter the desired octal operation
19     permissions.*/
20     printf("\nEnter the operation\n");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);

23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags                = 0\n");
27     printf("IPC_CREAT                    = 1\n");
28     printf("IPC_EXCL                      = 2\n");
29     printf("IPC_CREAT and IPC_EXCL        = 3\n");
30     printf("Flags                          = ");

31     /*Get the flag(s) to be set.*/
32     scanf("%d", &flags);

33     /*Check the values.*/
34     printf ("\nkey =0%x, opperm = 0%o, flags = 0%o\n",
```

(continued on next page)

```
35     key, opperm, flags);

36     /*Incorporate the control fields (flags) with
37     the operation permissions*/
38     switch (flags)
39     {
40     case 0: /*No flags are to be set.*/
41         opperm_flags = (opperm | 0);
42         break;
43     case 1: /*Set the IPC_CREAT flag.*/
44         opperm_flags = (opperm | IPC_CREAT);
45         break;
46     case 2: /*Set the IPC_EXCL flag.*/
47         opperm_flags = (opperm | IPC_EXCL);
48         break;
49     case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
50         opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51     }

52     /*Call the msgget system call.*/
53     msgqid = msgget (key, opperm_flags);

54     /*Perform the following if the call is unsuccessful.*/
55     if(msgqid == -1)
56     {
57         printf ("\nThe msgget system call failed!\n");
58         printf ("The error number = %d\n", errno);
59     }

60     /*Return the msgqid upon successful completion.*/
61     else
62         printf ("\nThe msgqid = %d\n", msgqid);
63     exit(0);
64 }
```

Controlling Message Queues

This section gives a detailed description of using the **msgctl** system call along with an example program which allows all of its capabilities to be exercised.

Using **msgctl**

The synopsis found in the **msgctl(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The **msgctl()** system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, it returns a -1 .

The **msqid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget()** system call.

The **cmd** argument can be replaced by one of the following control commands (flags):

IPC_STAT return the status information contained in the associated data structure for the specified **msqid**, and place it in the data structure pointed to by the * **buf** pointer in the user memory area.

Messages

- IPC_SET** for the specified **msqid**, set the effective user and group identification, operation permissions, and the number of bytes for the message queue.
- IPC_RMID** remove the specified **msqid** along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-5) is a menu driven program which allows all possible combinations of using the **msgctl(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out. This program begins (lines 5-9) by including the required header files as specified by the **msgctl(2)** entry in the *System V Reference Manual*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

uid	used to store the IPC_SET value for the effective user identification
gid	used to store the IPC_SET value for the effective group identification
mode	used to store the IPC_SET value for the operation permissions
bytes	used to store the IPC_SET value for the number of bytes in the message queue (msg_qbytes)
rtrn	used to store the return integer value from the system call
msqid	used to store and pass the message queue identifier to the system call
command	used to store the code for the desired control command so that subsequent processing can be performed on it
choice	used to determine which member is to be changed for the IPC_SET control command
msqid_ds	used to receive the specified message queue identifier's data structure when an IPC_STAT control command is performed
* buf	a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set

Note that the **msqid_ds** data structure in this program (line 16) uses the data structure located in the **msg.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

Messages

The next important thing to observe is that although the `*buf` pointer is declared to be a pointer to a data structure of the `msqid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier which is stored at the address of the `msqid` variable (lines 19, 20). This is required for every `msgctl` system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the `IPC_STAT` control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the `errno` variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

If the `IPC_SET` control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the

system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100-103), and the **msqid** along with its associated message queue and data structure are removed from the UNIX operating system. Note that the ***buf** pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **msgctl()** system call follows. It is suggested that the source program file be named **msgctl.c** and that the executable file be named **msgctl**.

Figure 8-5: **msgctl()** System Call Example

```

1  /*This is a program to illustrate
2  **the message control, msgctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode, bytes;
15     int rtrn, msqid, command, choice;
16     struct msqid_ds msqid_ds, *buf;
17     buf = &msqid_ds;

18     /*Get the msqid, and command.*/
19     printf("Enter the msqid = ");
20     scanf("%d", &msqid);

```

(continued on next page)

Messages

```
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");
23     printf("IPC_STAT   = 1\n");
24     printf("IPC_SET    = 2\n");
25     printf("IPC_RMID   = 3\n");
26     printf("Entry     = ");
27     scanf("%d", &command);

28     /*Check the values.*/
29     printf ("\nmsqid =%d, command = %d\n",
30            msqid, command);

31     switch (command)
32     {
33     case 1: /*Use msgctl() to duplicate
34            the data structure for
35            msqid in the msqid_ds area pointed
36            to by buf and then print it out.*/
37         rtn = msgctl(msqid, IPC_STAT,
38                    buf);
39         printf ("\nThe USER ID = %d\n",
40                buf->msg_perm.uid);
41         printf ("The GROUP ID = %d\n",
42                buf->msg_perm.gid);
43         printf ("The operation permissions = 0%o\n",
44                buf->msg_perm.mode);
45         printf ("The msg_qbytes = %d\n",
46                buf->msg_qbytes);
47         break;
48     case 2: /*Select and change the desired
49            member(s) of the data structure.*/
50         /*Get the original data for this msqid
51            data structure first.*/
52         rtn = msgctl(msqid, IPC_STAT, buf);
53         printf("\nEnter the number for the\n");
54         printf("member to be changed:\n");
55         printf("msg_perm.uid   = 1\n");
56         printf("msg_perm.gid   = 2\n");
57         printf("msg_perm.mode  = 3\n");
58         printf("msg_qbytes   = 4\n");
59         printf("Entry       = ");
```

(continued on next page)

```
60     scanf("%d", &choice);
61     /*Only one choice is allowed per
62     pass as an illegal entry will
63     cause repetitive failures until
64     msqid_ds is updated with
65     IPC_STAT.*/

66     switch(choice){
67     case 1:
68         printf("\nEnter USER ID = ");
69         scanf ("%d", &uid);
70         buf->msg_perm.uid = uid;
71         printf("\nUSER ID = %d\n",
72             buf->msg_perm.uid);
73         break;
74     case 2:
75         printf("\nEnter GROUP ID = ");
76         scanf("%d", &gid);
77         buf->msg_perm.gid = gid;
78         printf("\nGROUP ID = %d\n",
79             buf->msg_perm.gid);
80         break;
81     case 3:
82         printf("\nEnter MODE = ");
83         scanf("%o", &mode);
84         buf->msg_perm.mode = mode;
85         printf("\nMODE = 0%o\n",
86             buf->msg_perm.mode);
87         break;

88     case 4:
89         printf("\nEnter msg_bytes = ");
90         scanf("%d", &bytes);
91         buf->msg_qbytes = bytes;
92         printf("\nmsg_qbytes = %d\n",
93             buf->msg_qbytes);
94         break;
95     }

96     /*Do the change.*/
97     rtrn = msgctl(msqid, IPC_SET,
98         buf);
```

(continued on next page)

```

99         break;

100     case 3: /*Remove the msqid along with its
101             associated message queue
102             and data structure.*/
103         rtn = msgctl(msqid, IPC_RMID, NULL);
104     }
105     /*Perform the following if the call is unsuccessful.*/
106     if(rtn == -1)
107     {
108         printf ("\nThe msgctl system call failed!\n");
109         printf ("The error number = %d\n", errno);
110     }
111     /*Return the msqid upon successful completion.*/
112     else
113         printf ("\nMsgctl was successful for msqid = %d\n",
114             msqid);
115     exit (0);
116 }

```

Operations for Messages

This section gives a detailed description of using the **msgsnd(2)** and **msgrcv(2)** system calls, along with an example program which allows all of their capabilities to be exercised.

Using msgop

The synopsis found in the **msgop(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message

The **msgsnd** system call requires four arguments to be passed to it. It returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, **msgsnd()** returns a -1 .

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget()** system call.

The **msgp** argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The **msgsz** argument specifies the length of the character array in the data structure pointed to by the **msgp** argument. This is the length of the message. The maximum **size** of this array is determined by the MSGMAX system tunable parameter.

Messages

The **msg_qbytes** data structure member can be lowered from MSGMNB by using the **msgctl()** IPC_SET control command, but only the super-user can raise it afterwards.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if the total number of bytes allowed on the specified message queue are in use (**msg_qbytes** or MSGMNB), or the total system-wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a -1.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Receiving Messages

The **msgrcv()** system call requires five arguments to be passed to it, and it returns an integer value.

Upon successful completion, a value equal to the number of bytes received is returned and when unsuccessful it returns a -1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget()** system call.

The **msgp** argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The **msgsz** argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the **msgflg** argument.

The **msgtyp** argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the

lowest type that is less than or equal to its absolute value is received.

The **msgflg** argument allows the "blocking message operation" to be performed if the `IPC_NOWAIT` flag is not set (**msgflg** & `IPC_NOWAIT` = 0); this would occur if there is not a message on the message queue of the desired type (**msgtyp**) to be received. If the `IPC_NOWAIT` flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. `Msgflg` can also specify that the system call fail if the message is longer than the **size** to be received; this is done by not setting the `MSG_NOERROR` flag in the **msgflg** argument (**msgflg** & `MSG_NOERROR` = 0). If the `MSG_NOERROR` flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv()**.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-6) is a menu driven program which allows all possible combinations of using the **msgsnd()** and **msgrcv(2)** system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgop(2)** entry in the *System V Reference Manual*. Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Messages

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- sndbuf** used as a buffer to contain a message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13) The **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message **size** (**MSGMAX**) for the Computer where in **msgbuf** it is set to one (1) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.
- rcvbuf** used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)
- * msgp** used as a pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers
- i** used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd()** system call; it is also used as a counter to output the received message for the **msgrcv()** system call
- c** used to receive the input character from the **getchar()** function (line 50)
- flag** used to store the code of **IPC_NOWAIT** for the **msgsnd()** system call (line 61)

flags	used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the msgrcv() system call (line 117)
choice	used to store the code for sending or receiving (line 30)
rtrn	used to store the return values from all system calls
msqid	used to store and pass the desired message queue identifier for both system calls
msgsz	used to store and pass the size of the message to be sent or received
msgflg	used to pass the value of flag for sending or the value of flags for receiving
msgtyp	used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a **msqid_ds** data structure is set up in the program (line 21) with a pointer which is initialized to point to it (line 22); this will allow the data structure members that are affected by message operations to be observed. They are observed by using the **msgctl()** (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored at the address of the choice variable (lines 23-30). Depending upon the code, the program proceeds as in the following **msgsnd** or **msgrcv** sections.

msgsnd

When the code is to send a message, the **msgp** pointer is initialized (line 33) to the address of the send data structure, **sndbuf**. Next, a message type must be entered for the message; it is stored at the address of the variable **msgtyp** (line 42), and then (line 43) it is put

Messages

into the `mtype` member of the data structure pointed to by `msgp`.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the `mtext` array of the data structure (lines 48-51). This will continue until an end of file is recognized which for the `getchar()` function is a control-d (CTRL-D) immediately following a carriage return (<CR>). When this happens, the `size` of the message is determined by adding one to the `i` counter (lines 52, 53) as it stored the message beginning in the zero array element of `mtext`. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of `msgsz`.

The message is immediately echoed from the `mtext` array of the `sndbuf` data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the `IPC_NOWAIT` flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, `IPC_NOWAIT` is logically ORed with `msgflg`; otherwise, `msgflg` is set to zero.

The `msgsnd()` system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed which should be zero (lines 73-76).

Every time a message is successfully sent, there are three members of the associated data structure which are updated. They are described as follows:

- msg_qnum** represents the total number of messages on the message queue; it is incremented by one.
- msg_lspid** contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.

msg_stime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The **msgp** pointer is initialized to the **rcvbuf** data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested, and it is stored at the address of **msqid** (lines 100-103).

The message type is requested, and it is stored at the address of **msgtyp** (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored at the address of flags (lines 108-117). Depending upon the selected combination, **msgflg** is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested, and it is stored at the address of **msgsz** (lines 134-137).

The **msgrcv()** system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, there are three members of the associated data structure which are updated; they are described as follows:

Messages

- msg_qnum** contains the number of messages on the message queue; it is decremented by one.
- msg_lrpid** contains the process identification (PID) of the last process receiving a message; it is set accordingly.
- msg_rtime** contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The example program for the **msgop(2)** system calls follows. It is suggested that the program be put into a source file called **msgop.c** and then into an executable file called **msgop**.

Figure 8-6: **msgop(2)** System Call Example

```

1  /*This is a program to illustrate
2  **the message operations, msgop(2),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10 struct msgbuf1 {
11     long   mtype;
12     char   mtext[8192];
13 } sndbuf, rcvbuf, *msgp;

14 /*Start of main C language program*/
15 main()
16 {
17     extern int errno;
18     int i, c, flag, flags, choice;
19     int rtn, msgqid, msgsz, msgflg;
20     long mtype, msgtyp;

```

(continued on next page)

```
21     struct msqid_ds msqid_ds, *buf;
22     buf = &msqid_ds;

23     /*Select the desired operation.*/
24     printf("Enter the corresponding\n");
25     printf("code to send or\n");
26     printf("receive a message:\n");
27     printf("Send           = 1\n");
28     printf("Receive          = 2\n");
29     printf("Entry            = ");
30     scanf("%d", &choice);

31     if(choice == 1) /*Send a message.*/
32     {
33         msgp = &sndbuf; /*Point to user send structure.*/

34         printf("\nEnter the msqid of\n");
35         printf("the message queue to\n");
36         printf("handle the message = ");
37         scanf("%d", &msqid);

38         /*Set the message type.*/
39         printf("\nEnter a positive integer\n");
40         printf("message type (long) for the\n");
41         printf("message = ");
42         scanf("%d", &msgtyp);
43         msgp->mtype = msgtyp;

44         /*Enter the message to send.*/
45         printf("\nEnter a message: \n");

46         /*A control-d (^d) terminates as
47         EOF.*/

48         /*Get each character of the message
49         and put it in the mtext array.*/
50         for(i = 0; ((c = getchar()) != EOF); i++)
51             sndbuf.mtext[i] = c;

52         /*Determine the message size.*/
53         msgsz = i + 1;
```

(continued on next page)

Messages

```
54      /*Echo the message to send.*/
55      for(i = 0; i < msgsz; i++)
56          putchar(sndbuf.mtext[i]);

57      /*Set the IPC_NOWAIT flag if
58      desired.*/
59      printf("\nEnter a 1 if you want the\n");
60      printf("the IPC_NOWAIT flag set: ");
61      scanf("%d", &flag);
62      if(flag == 1)
63          msgflg |= IPC_NOWAIT;
64      else
65          msgflg = 0;

66      /*Check the msgflg.*/
67      printf("\nmsgflg = 0%o\n", msgflg);

68      /*Send the message.*/
69      rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
70      if(rtrn == -1)
71          printf("\nMsgsnd failed. Error = %d\n",
72              errno);
73      else {
74          /*Print the value of test which
75          should be zero for successful.*/
76          printf("\nValue returned = %d\n", rtrn);

77          /*Print the size of the message
78          sent.*/
79          printf("\nMsgsz = %d\n", msgsz);

80          /*Check the data structure update.*/
81          msgctl(msqid, IPC_STAT, buf);

82          /*Print out the affected members.*/

83          /*Print the incremented number of
84          messages on the queue.*/
85          printf("\nThe msg_qnum = %d\n",
86              buf->msg_qnum);
87          /*Print the process id of the last sender.*/
88          printf("The msg_lspid = %d\n",
```

(continued on next page)

```
89         buf->msg_lspid);
90         /*Print the last send time.*/
91         printf("The msg_stime = %d\n",
92             buf->msg_stime);
93     }
94 }

95 if(choice == 2) /*Receive a message.*/
96 {
97     /*Initialize the message pointer
98     to the receive buffer.*/
99     msgp = &rcvbuf;

100     /*Specify the message queue which contains
101     the desired message.*/
102     printf("\nEnter the msgqid = ");
103     scanf("%d", &msgqid);

104     /*Specify the specific message on the queue
105     by using its type.*/
106     printf("\nEnter the msgtyp = ");
107     scanf("%d", &msgtyp);

108     /*Configure the control flags for the
109     desired actions.*/
110     printf("\nEnter the corresponding code\n");
111     printf("to select the desired flags: \n");
112     printf("No flags           = 0\n");
113     printf("MSG_NOERROR             = 1\n");
114     printf("IPC_NOWAIT                = 2\n");
115     printf("MSG_NOERROR and IPC_NOWAIT = 3\n");
116     printf("Flags                     = ");
117     scanf("%d", &flags);

118     switch(flags) {
119         /*Set msgflg by ORing it with the appropriate
120         flags (constants).*/
121     case 0:
122         msgflg = 0;
123         break;
124     case 1:
125         msgflg |= MSG_NOERROR;
```

(continued on next page)

Messages

```

126         break;
127     case 2:
128         msgflg |= IPC_NOWAIT;
129         break;
130     case 3:
131         msgflg |= MSG_NOERROR | IPC_NOWAIT;
132         break;
133     }

134     /*Specify the number of bytes to receive.*/
135     printf("\nEnter the number of bytes\n");
136     printf("to receive (msgsz) = ");
137     scanf("%d", &msgsz);

138     /*Check the values for the arguments.*/
139     printf("\nmsgid = %d\n", msgid);
140     printf("\nmsgtyp = %d\n", msgtyp);
141     printf("\nmsgsz = %d\n", msgsz);
142     printf("\nmsgflg = 0x%o\n", msgflg);

143     /*Call msgrcv to receive the message.*/
144     rtrn = msgrcv(msgid, msgp, msgsz, msgtyp, msgflg);

145     if(rtrn == -1) {
146         printf("\nMsgrcv failed. ");
147         printf("Error = %d\n", errno);
148     }
149     else {
150         printf ("\nMsgctl was successful\n");
151         printf("for msgid = %d\n",
152             msgid);

153         /*Print the number of bytes received,
154            it is equal to the return
155            value.*/
156         printf("Bytes received = %d\n", rtrn);

157         /*Print the received message.*/
158         for(i = 0; i<=rtrn; i++)
159             putchar(rcvbuf.mtext[i]);
160     }
161     /*Check the associated data structure.*/

```

(continued on next page)


```
162         msgctl(msgid, IPC_STAT, buf);
163         /*Print the decremented number of messages.*/
164         printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165         /*Print the process id of the last receiver.*/
166         printf("The msg_lrpid = %d\n", buf->msg_lrpid);
167         /*Print the last message receive time*/
168         printf("The msg_rtime = %d\n", buf->msg_rtime);
169     }
170 }
```

Messages

This page is intentionally left blank

Semaphores

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the UNIX operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, SEMMSL has a default value of 25. Semaphore sets are created by using the **semget(2)** system call.

The process performing the **semget(2)** system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl(2)**, semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl(2)** to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore(s). Each semaphore within a set can be manipulated in two ways with the **semop(2)** system call (which is documented in the *System V Reference Manual*):

- incremented
- decremented

To increment a semaphore, an integer value of the desired magnitude is passed to the **semop(2)** system call. To decrement a semaphore, a minus (-) value of the desired magnitude is passed.

The UNIX operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

Semaphores

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process which is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is accomplished by passing a value of zero to the **semop(2)** system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "non-blocking semaphore operation." In this case, the process is returned a known error code (-1), and the external **errno** variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop(2)**, semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done

successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed or one "nonblocking operation" is unsuccessful and none are changed. All of this is commonly referred to as being "atomically performed."

The ability to undo operations requires the UNIX operating system to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation which is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Semaphores

Using Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); it is used to identify or reference a particular data structure and semaphore set.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (**nsems**) in a semaphore set is user selectable. The following members are in each structure within a semaphore set:

- semaphore text map address
- process identification (PID) performing last operation
- number of processes awaiting the semaphore value to become greater than its current value
- number of processes awaiting the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- operation permissions data (operation permissions structure)
- pointer to first semaphore in the set (array)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The C Programming Language data structure definition for the semaphore set (array member) is as follows:

```

struct sem
{
    ushort  semval;      /* semaphore text map address */
    short   sempid;     /* pid of last operation */
    ushort  semncnt;    /* # awaiting semval > cval */
    ushort  semzcnt;    /* # awaiting semval = 0 */
};

```

It is located in the **#include <sys/sem.h>** header file.

Likewise, the structure definition for the associated semaphore data structure is as follows:

```

struct semid_ds
{
    struct ipc_perm sem_perm; /* operation permission struct */
    struct sem      *sem_base; /* ptr to first semaphore in set */
    ushort         sem_nsems; /* # of semaphores in set */
    time_t         sem_otime; /* last semop time */
    time_t         sem_ctime; /* last change time */
};

```

It is also located in the **#include <sys/sem.h>** header file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 8-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the "Messages" section.

The **semget(2)** system call is used to perform two tasks when only the **IPC_CREAT** flag is set in the **semflg** argument that it receives:

Semaphores

- to get a new **semid** and create an associated data structure and semaphore set for it
- to return an existing **semid** that already has an associated data structure and semaphore set

The task performed is determined by the value of the **key** argument passed to the **semget(2)** system call. For the first task, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set created for it provided no system tunable parameter would be exceeded.

There is also a provision for specifying a **key** of value zero (0) which is known as the private **key** (**IPC_PRIVATE** = 0); when specified, a new **semid** is always returned with an associated data structure and semaphore set created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the **KEY** field for the **semid** is all zeros.

When performing the first task, the process which calls **semget()** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Semaphores" section in this chapter. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If it is not desired to have an existing **semid** returned, a control command (**IPC_EXCL**) can be specified (set) in the **semflg** argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for **nsems**. The details of using this system call are discussed in the "Using **semget**" section of this chapter.

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop**(2)] and semaphore control [**semctl**(2)] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to perform these operations. It is called **semop**(2). Refer to the "Operations on Semaphores" section in this chapter for details of this system call.

Semaphore control is done by using the **semctl**(2) system call. These control operations permit you to control the semaphore facility in the following ways:

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero
- to get all semaphore values in a set and place them in an array in user memory
- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values, status, of a semaphore set into user memory area
- to change operation permissions for a semaphore set
- to remove a particular **semid** from the UNIX operating system along with its associated data structure and semaphore set

Semaphores

Refer to the "Controlling Semaphores" section in this chapter for details of the **semctl(2)** system call.

Getting Semaphores

This section contains a detailed description of using the **semget(2)** system call along with an example program illustrating its use.

Using semget

The synopsis found in the **semget(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semg)
key_t key;
int nsems, semg;
```

The following line in the synopsis:

```
int semget (key, nsems, semflg)
```

informs you that **semget()** is a function with three formal arguments that returns an integer type value, upon successful completion (**semid**). The next two lines:

```
key_t key;
int nsems, semflg;
```

declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier (**semid**) that was discussed above. As declared, the process calling the **semget()** system call must supply three actual arguments to be passed to the formal **key**, **nsems**, and **semflg** arguments.

A new **semid** with an associated semaphore set and data structure is provided if either

- **key** is equal to `IPC_PRIVATE`,
- or**
- **key** is passed a unique hexadecimal integer, and **semflg** ANDed with `IPC_CREAT` is `TRUE`.

The value passed to the **semflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/alter attributes and execution modes determine the user/group/other attributes of the **semflg** argument. They are collectively referred to as "operation permissions." Figure 8-7 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

Figure 8-7: Operation Permissions Codes

Semaphores

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants **#define**'d in the **sem.h** header file which can be used for the user (OWNER). They are as follows:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Figure 8-8 contains the names of the constants which apply to the **semget(2)** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 8-8: Control Commands (Flags)

The value for **semflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
Read by User	=	0 0 4 0 0	0 000 000 100 000 000
semflg	=	0 1 4 0 0	0 000 001 100 000 000

420

The **semflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
msgid = msgget (key, (IPC_CREAT | 0400));
```

```
msgid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **semget(2)** entry in the *System V Reference Manual*, success or failure of this system call depends upon the actual argument values for **key**, **nsems**, **semflg** or system tunable parameters. The system call will attempt to return a new **semid** if one of the following conditions is true:

- Key is equal to **IPC_PRIVATE** (0)
- Key does not already have a **semid** associated with it, and (**semflg & IPC_CREAT**) is "true" (not zero).

The **key** argument can be set to **IPC_PRIVATE** in the following ways:

```
msgid = msgget (IPC_PRIVATE, msgflg);
```

or

```
msgid = msgget ( 0, msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

Exceeding the **SEMMNI**, **SEMMNS**, or **SEMMSL** system tunable parameters will always cause a failure. The **SEMMNI** system tunable parameter determines the maximum number of unique semaphore sets (**semid**'s) in the UNIX operating system. The **SEMMNS** system tunable parameter determines the maximum number of semaphores in all semaphore sets system wide. The **SEMMSL** system tunable parameter determines the maximum number of semaphores in each semaphore set.

Semaphores

The second condition is satisfied if the value for **key** is not already associated with a **semid**, and the bitwise ANDing of **semflg** and **IPC_CREAT** is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the **IPC_CREAT** flag is set (**semflg** | **IPC_CREAT**). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```

msgflg   = x 1 x x x (x = immaterial)
& IPC_CREAT = 0 1 0 0 0
result   = 0 1 0 0 0 (not zero)

```

Since the result is not zero, the flag is set or "true." **SEMMNI**, **SEMMNS**, and **SEMMSL** apply here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with **IPC_CREAT** to exclusively have the system call fail if, and only if, a **semid** exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. In other words, when both **IPC_CREAT** and **IPC_EXCL** are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if the key equals zero (**IPC_PRIVATE**) and no system tunable parameters are exceeded.

Refer to the **semget(2)** manual page for specific associated data structure initialization for successful completion.

Example Program

The example program in this section (Figure 8-9) is a menu driven program which allows all possible combinations of using the **semget(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **semget(2)** entry in the *System V Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **semflg** argument
- **semid**—used for returning the semaphore set identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of **nsems**.

Semaphores

The system call is made next, and the result is stored at the address of the **semid** variable (lines 60, 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (-1), it is tested to see if an error occurred (line 63). If **semid** equals -1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget(2)** system call follows. It is suggested that the source program file be named **semget.c** and that the executable file be named **semget**.

Figure 8-9: **semget()** System Call Example

```
1  /*This is a program to illustrate
2  **the semaphore get, semget(),
3  **system call capabilities.*/

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key; /*declare as long integer*/
13     int opperm, flags, nsems;
14     int semid, opperm_flags;

15     /*Enter the desired key*/
16     printf("\nEnter the desired key in hex = ");
```

(continued on next page)


```
17     scanf("%x", &key);

18     /*Enter the desired octal operation
19     permissions.*/
20     printf("\nEnter the operation\n");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);

23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags           = 0\n");
27     printf("IPC_CREAT             = 1\n");
28     printf("IPC_EXCL              = 2\n");
29     printf("IPC_CREAT and IPC_EXCL  = 3\n");
30     printf("Flags                   = ");
31     /*Get the flags to be set.*/
32     scanf("%d", &flags);

33     /*Error checking (debugging)*/
34     printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35     key, opperm, flags);
36     /*Incorporate the control fields (flags) with
37     the operation permissions.*/
38     switch (flags)
39     {
40     case 0: /*No flags are to be set.*/
41         opperm_flags = (opperm | 0);
42         break;
43     case 1: /*Set the IPC_CREAT flag.*/
44         opperm_flags = (opperm | IPC_CREAT);
45         break;
46     case 2: /*Set the IPC_EXCL flag.*/
47         opperm_flags = (opperm | IPC_EXCL);
48         break;
49     case 3: /*Set the IPC_CREAT and IPC_EXCL
50             flags.*/
51         opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52     }

53     /*Get the number of semaphores for this set.*/
54     printf("\nEnter the number of\n");
```

(continued on next page)

Semaphores

```
55     printf("desired semaphores for\n");
56     printf("this set (25 max) = ");
57     scanf("%d", &nsems);

58     /*Check the entry.*/
59     printf("\nNsems = %d\n", nsems);

60     /*Call the semget system call.*/
61     semid = semget(key, nsems, opperm_flags);

62     /*Perform the following if the call is unsuccessful.*/
63     if(semid == -1)
64     {
65         printf("The semget system call failed!\n");
66         printf("The error number = %d\n", errno);
67     }
68     /*Return the semid upon successful completion.*/
69     else
70         printf("\nThe semid = %d\n", semid);
71     exit(0);
72 }
```

426

Controlling Semaphores

This section contains a detailed description of using the **semctl(2)** system call along with an example program which allows all of its capabilities to be exercised.

Using semctl

The synopsis found in the **semctl(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
    int val;
    struct semid_ds *bu;
    ushort array[];
} arg;
```

The **semctl(2)** system call requires four arguments to be passed to it, and it returns an integer value.

The **semid** argument must be a valid, non-negative, integer value that has already been created by using the **semget(2)** system call.

The **semnum** argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The **cmd** argument can be replaced by one of the following control commands (flags):

- **GETVAL**—return the value of a single semaphore within a semaphore set

Semaphores

- SETVAL—set the value of a single semaphore within a semaphore set
- GETPID—return the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set
- GETNCNT—return the number of processes waiting for the value of a particular semaphore to become greater than its current value
- GETZCNT—return the number of processes waiting for the value of a particular semaphore to be equal to zero
- GETALL—return the values for all semaphores in a semaphore set
- SETALL—set all semaphore values in a semaphore set
- IPC_STAT—return the status information contained in the associated data structure for the specified **semid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area; **arg.buf** is the union member that contains the value of **buf**
- IPC_SET—for the specified semaphore set (**semid**), set the effective user/group identification and operation permissions
- IPC_RMID—remove the specified (**semid**) semaphore set along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

The **arg** argument is used to pass the system call the appropriate union member for the control command to be performed:

- **arg.val**
- **arg.buf**
- **arg.array**

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **semget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-10) is a menu driven program which allows all possible combinations of using the **semctl(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **semctl(2)** entry in the *System V Reference Manual*. Note that in this program **errno** is declared as an external variable, and therefore the **errno.h** header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. Those declared for this program and their purpose are as follows:

- **semid_ds**—used to receive the specified semaphore set identifier's data structure when an **IPC_STAT** control command is performed

Semaphores

- **c**—used to receive the input values from the **scanf(3S)** function, (line 117) when performing a **SETALL** control command
- **i**—used as a counter to increment through the union **arg.array** when displaying the semaphore values for a **GETALL** (lines 97-99) control command, and when initializing the **arg.array** when performing a **SETALL** (lines 115-119) control command
- **length**—used as a variable to test for the number of semaphores in a set against the **i** counter variable (lines 97, 115)
- **uid**—used to store the **IPC_SET** value for the effective user identification
- **gid**—used to store the **IPC_SET** value for the effective group identification
- **mode**—used to store the **IPC_SET** value for the operation permissions
- **rtrn**—used to store the return integer from the system call which depends upon the control command or a **-1** when unsuccessful
- **semid**—used to store and pass the semaphore set identifier to the system call
- **semnum**—used to store and pass the semaphore number to the system call
- **cmd**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member (**uid**, **gid**, **mode**) for the **IPC_SET** control command that is to be changed
- **arg.val**—used to pass the system call a value to set (**SETVAL**) or to store (**GETVAL**) a value returned from the system call for a single semaphore (union member)

- **arg.buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set (union member)
- **arg.array**—used to store the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

Note that the **semid_ds** data structure in this program (line 14) uses the data structure located in the **sem.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The **arg** union (lines 18-22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24). This number corresponds to the maximum number of semaphores allowed per set (SEMMSL), a system tunable parameter.

The next important program aspect to observe is that although the ***buf** pointer member (**arg.buf**) of the union is declared to be a pointer to a data structure of the **semid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now that all of the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored at the address of the **semid** variable (lines 25-27). This is required for all **semctl(2)** system calls.

Semaphores

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored at the address of the **cmd** variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored at the address of the **semnum** variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored at the address of the **semnum** variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the **arg.val** member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the **semnum** variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the **semnum** variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91). Next, the system call is made and, upon success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is

Semaphores

unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the **IPC_SET** control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl(2)** system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the choice variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as for **GETVAL** above.

If the **IPC_RMID** control command (code 10) is selected, the system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the UNIX operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl(2)** system call follows. It is suggested that the source program file be named **semctl.c** and that the executable file be named **semctl**.

Figure 8-10: `semctl(2)` System Call Example

```

1  /*This is a program to illustrate
2  **the semaphore control, semctl(2),
3  **system call capabilities.
4  */
5
6  /*Include necessary header files.*/
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11
12 /*Start of main C language program*/
13 main()
14 {
15     extern int errno;
16     struct semid_ds semid_ds;
17     int c, i, length;
18     int uid, gid, mode;
19     int retn, semid, semnum, cmd, choice;
20     union semun {
21         int val;
22         struct semid_ds *buf;
23         ushort array[25];
24     } arg;
25
26     /*Initialize the data structure pointer.*/
27     arg.buf = &semid_ds;
28
29     /*Enter the semaphore ID.*/
30     printf("Enter the semid = ");
31     scanf("%d", &semid);
32
33     /*Choose the desired command.*/
34     printf("\nEnter the number for\n");
35     printf("the desired cmd:\n");
36     printf("GETVAL      = 1\n");
37     printf("SETVAL      = 2\n");
38     printf("GETPID      = 3\n");
39     printf("GETNCNT     = 4\n");
40     printf("GETZCNT     = 5\n");

```

(continued on next page)

Semaphores

```
36     printf("GETALL      = 6\n");
37     printf("SETALL      = 7\n");
38     printf("IPC_STAT    = 8\n");
39     printf("IPC_SET     = 9\n");
40     printf("IPC_RMID    = 10\n");
41     printf("Entry      = ");
42     scanf("%d", &cmd);

43     /*Check entries.*/
44     printf ("\nsemid =%d, cmd = %d\n\n",
45            semid, cmd);

46     /*Set the command and do the call.*/
47     switch (cmd)
48     {

49     case 1: /*Get a specified value.*/
50         printf("\nEnter the semnum = ");
51         scanf("%d", &semnum);
52         /*Do the system call.*/
53         retrn = semctl(semid, semnum, GETVAL, 0);
54         printf("\nThe semval = %d\n", retrn);
55         break;

56     case 2: /*Set a specified value.*/
57         printf("\nEnter the semnum = ");
58         scanf("%d", &semnum);
59         printf("\nEnter the value = ");
60         scanf("%d", &arg.val);
61         /*Do the system call.*/
62         retrn = semctl(semid, semnum, SETVAL, arg.val);
63         break;

64     case 3: /*Get the process ID.*/
65         retrn = semctl(semid, 0, GETPID, 0);
66         printf("\nThe sempid = %d\n", retrn);
67         break;

68     case 4: /*Get the number of processes
69             waiting for the semaphore to
70             become greater than its current
71             value.*/
72         printf("\nEnter the semnum = ");
73         scanf("%d", &semnum);
74         /*Do the system call.*/
```

(continued on next page)

```

75         retrn = semctl(semid, semnum, GETNCNT, 0);
76         printf("\n\nThe semncnt = %d", retrn);
77         break;

78     case 5: /*Get the number of processes
79             waiting for the semaphore
80             value to become zero.*/
81         printf("\n\nEnter the semnum = ");
82         scanf("%d", &semnum);
83         /*Do the system call.*/
84         retrn = semctl(semid, semnum, GETZCNT, 0);
85         printf("\n\nThe semzcnt = %d", retrn);
86         break;

87     case 6: /*Get all of the semaphores.*/
88         /*Get the number of semaphores in
89             the semaphore set.*/
90         retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91         length = arg.buf->sem_nsems;
92         if(retrn == -1)
93             goto ERROR;
94         /*Get and print all semaphores in the
95             specified set.*/
96         retrn = semctl(semid, 0, GETALL, arg.array);
97         for (i = 0; i < length; i++)
98             {
99                 printf("%d", arg.array[i]);
100                /*Seperate each
101                semaphore.*/
102                printf("%c", ' ');
103            }
104         break;

105     case 7: /*Set all semaphores in the set.*/
106         /*Get the number of semaphores in
107             the set.*/
108         retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109         length = arg.buf->sem_nsems;
110         printf("Length = %d\n", length);
111         if(retrn == -1)
112             goto ERROR;
113         /*Set the semaphore set values.*/

```

(continued on next page)

Semaphores

```
114     printf("\nEnter each value:\n");
115     for(i = 0; i < length ; i++)
116     {
117         scanf("%d", &c);
118         arg.array[i] = c;
119     }
120     /*Do the system call.*/
121     retrn = semctl(semid, 0, SETALL, arg.array);
122     break;

123     case 8: /*Get the status for the semaphore set.*/
124             /*Get and print the current status values.*/
125             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
126             printf ("\nThe USER ID = %d\n",
127                     arg.buf->sem_perm.uid);
128             printf ("The GROUP ID = %d\n",
129                     arg.buf->sem_perm.gid);
130             printf ("The operation permissions = 0%o\n",
131                     arg.buf->sem_perm.mode);
132             printf ("The number of semaphores in set = %d\n",
133                     arg.buf->sem_nsems);
134             printf ("The last semop time = %d\n",
135                     arg.buf->sem_otime);

136             printf ("The last change time = %d\n",
137                     arg.buf->sem_ctime);
138             break;

139     case 9: /*Select and change the desired
140             member of the data structure.*/
141             /*Get the current status values.*/
142             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
143             if(retrn == -1)
144                 goto ERROR;
145             /*Select the member to change.*/
146             printf("\nEnter the number for the\n");
147             printf("member to be changed:\n");
148             printf("sem_perm.uid   = 1\n");
149             printf("sem_perm.gid   = 2\n");
150             printf("sem_perm.mode  = 3\n");
151             printf("Entry         = ");
152             scanf("%d", &choice);
```

(continued on next page)

```

155         switch(choice){
156             case 1: /*Change the user ID.*/
157                 printf("\nEnter USER ID = ");
158                 scanf("%d", &uid);
159                 arg.buf->sem_perm.uid = uid;
160                 printf("\nUSER ID = %d\n",
161                     arg.buf->sem_perm.uid);
162                 break;
163             case 2: /*Change the group ID.*/
164                 printf("\nEnter GROUP ID = ");
165                 scanf("%d", &gid);
166                 arg.buf->sem_perm.gid = gid;
167                 printf("\nGROUP ID = %d\n",
168                     arg.buf->sem_perm.gid);
169                 break;
170             case 3: /*Change the mode portion of
171                 the operation
172                 permissions.*/
173                 printf("\nEnter MODE = ");
174                 scanf("%o", &mode);
175                 arg.buf->sem_perm.mode = mode;
176                 printf("\nMODE = 0%o\n",
177                     arg.buf->sem_perm.mode);
178                 break;
179             }
180             /*Do the change.*/
181             retrn = semctl(semid, 0, IPC_SET, arg.buf);
182             break;
183             case 10: /*Remove the semid along with its
184                 data structure.*/
185                 retrn = semctl(semid, 0, IPC_RMID, 0);
186             }
187             /*Perform the following if the call is unsuccessful.*/
188             if(retrn == -1)
189             {
190                 ERROR:
191                 printf ("\n\nThe semctl system call failed!\n");
192                 printf ("The error number = %d\n", errno);
193                 exit(0);

```

(continued on next page)

Semaphores

```
194     }
195     printf ("\n\nThe semctl system call was successful\n");
196     printf ("for semid = %d\n", semid);
197     exit (0);
198 }
```

Operations on Semaphores

This section contains a detailed description of using the **semop(2)** system call along with an example program which allows all of its capabilities to be exercised.

Using semop

The synopsis found in the **semop(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```


The **semop(2)** system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned and when unsuccessful it returns a `-1`.

The **semid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **semget(2)** system call.

The **sops** argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number
- the operation to be performed
- the control command (flags)

The ****sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. **sembuf** is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file.

The **nsops** argument specifies the length of the array (the number of structures in the array). The maximum **size** of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each **semop(2)** system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value

Semaphores

- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- **IPC_NOWAIT**—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which **IPC_NOWAIT** is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
- **SEM_UNDO**—this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the **IPC_NOWAIT** flag set. That is, the blocked operation waits until it can perform its operation; and when it and all succeeding operations are successful, all operations with the **SEM_UNDO** flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

Example Program

The example program in this section (Figure 9-11) is a menu driven program which allows all possible combinations of using the **semop(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop(2)** entry in the *System V Reference Manual*. Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since the declarations are local to the program. The variables declared for this program and their purpose are as follows:

- **sembuf[10]**—used as an array buffer (line 14) to contain a maximum of ten **sembuf** type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each **semop(2)** system call
- ***sops**—used as a pointer (line 14) to **sembuf[10]** for the system call and for accessing the structure members within the array
- **rtrn**—used to store the return values from the system call
- **flags**—used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the **semop(2)** system call (line 60)
- **i**—used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79)
- **nsops**—used to specify the number of semaphore operations for the system call—must be less than or equal to SEMOPM
- **semid**—used to store the desired semaphore set identifier for the system call

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). Semid is stored at the address of the **semid** variable (line 23).

Semaphores

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**. **sops** is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86, 87). **sembuf** could be used directly, if desired, instead of **sops** in the system call. The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl(2)** GETALL control command.

The example program for the **semop(2)** system call follows. It is suggested that the source program file be named **semop.c** and that the executable file be named **semop**.

Figure 8-11: **semop(2)** System Call Example

```
1  /*This is a program to illustrate
2  **the semaphore operations, semop(2),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     struct sembuf sembuf[10], *sops;
15     char string[];
16     int retrn, flags, sem_num, i, semid;
17     unsigned nsops;
18     sops = sembuf; /*Pointer to array sembuf.*/

19     /*Enter the semaphore ID.*/
20     printf("\nEnter the semid of\n");
21     printf("the semaphore set to\n");
22     printf("be operated on = ");
23     scanf("%d", &semid);
24     printf("\nsemid = %d", semid);

25     /*Enter the number of operations.*/
26     printf("\nEnter the number of semaphore\n");
27     printf("operations for this set = ");
28     scanf("%d", &nsops);
29     printf("\nnsops = %d", nsops);

30     /*Initialize the array for the
31     number of operations to be performed.*/
32     for(i = 0; i < nsops; i++, sops++)
33     {

34         /*This determines the semaphore in
35         the semaphore set.*/
```

(continued on next page)

Semaphores

```
36     printf("\nEnter the semaphore\n");
37     printf("number (sem_num) = ");
38     scanf("%d", &sem_num);
39     sops->sem_num = sem_num;
40     printf("\nThe sem_num = %d", sops->sem_num);

41     /*Enter a (-)number to decrement,
42     an unsigned number (no +) to increment,
43     or zero to test for zero. These values
44     are entered into a string and converted
45     to integer values.*/
46     printf("\nEnter the operation for\n");
47     printf("the semaphore (sem_op) = ");
48     scanf("%s", string);
49     sops->sem_op = atoi(string);
50     printf("\nsem_op = %d\n", sops->sem_op);

51     /*Specify the desired flags.*/
52     printf("\nEnter the corresponding\n");
53     printf("number for the desired\n");
54     printf("flags:\n");
55     printf("No flags           = 0\n");
56     printf("IPC_NOWAIT              = 1\n");
57     printf("SEM_UNDO                 = 2\n");
58     printf("IPC_NOWAIT and SEM_UNDO = 3\n");
59     printf("          Flags          = ");
60     scanf("%d", &flags);

61     switch(flags)
62     {
63     case 0:
64         sops->sem_flg = 0;
65         break;
66     case 1:
67         sops->sem_flg = IPC_NOWAIT;
68         break;
69     case 2:
70         sops->sem_flg = SEM_UNDO;
71         break;
72     case 3:
73         sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74         break;
```

(continued on next page)

```
75     }
76     printf("\nFlags = %o\n", sops->sem_flg);
77 }

78 /*Print out each structure in the array.*/
79 for(i = 0; i < nsops; i++)
80 {
81     printf("\nsem_num = %d\n", sembuf[i].sem_num);
82     printf("sem_op = %d\n", sembuf[i].sem_op);
83     printf("sem_flg = %o\n", sembuf[i].sem_flg);
84     printf("%c", ' ');
85 }

86 sops = sembuf; /*Reset the pointer to
87                sembuf[0].*/

88 /*Do the semop system call.*/
89 retn = semop(semid, sops, nsops);
90 if(retn == -1) {
91     printf("\nSemop failed. ");
92     printf("Error = %d\n", errno);
93 }
94 else {
95     printf ("\nSemop was successful\n");
96     printf("for semid = %d\n", semid);

97     printf("Value returned = %d\n", retn);
98 }
99 }
```

This page is intentionally left blank

Shared Memory

The shared memory type of IPC allows two or more processes (executing programs) to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware dependent.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget(2)** system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- **shmat(2)** – shared memory attach
- **shmdt(2)** – shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl(2)** system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment

Shared Memory

using the **shmctl(2)** system call.

System calls, which are documented in the *System V Reference Manual*, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the UNIX operating system at any point in time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (**shmid**); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size
- segment descriptor
- process identification performing last operation
- process identification of creator
- current number of processes attached

- in memory number of processes attached
- last attach time
- last detach time
- last change time

The C Programming Language data structure definition for the shared memory segment data structure is located in the `/usr/include/sys/shm.h` header file. It is as follows:

```

/*
**  There is a shared mem id data structure for
**  each segment in the system.
*/

struct shm_id {
    struct ipc_perm    shm_perm;    /* operation permission struct */
    int                shm_segsz;   /* segment size */
    struct region      *shm_reg;    /* ptr to region structure */
    char               pad[4];      /* for swap compatibility */
    ushort             shm_lpid;    /* pid of last shmop */
    ushort             shm_cpid;    /* pid of creator */
    ushort             shm_nattch;  /* used only for shminfo */
    ushort             shm_cnattch; /* used only for shminfo */
    time_t             shm_atime;   /* last shmat time */
    time_t             shm_dtime;   /* last shmdt time */
    time_t             shm_ctime;   /* last change time */
};

```

Note that the `shm_perm` member of this structure uses `ipc_perm` as a template. The breakout for the operation permissions data structure is shown in Figure 8-1.

The `ipc_perm` data structure is the same for all IPC facilities, and it is located in the `#include <sys/ipc.h>` header file. It is shown in the introduction section of "Messages."

Shared Memory

Figure 8-12 is a table that shows the shared memory state information.

Lock Bit	Swap Bit	Allocated Bit	Implied State
0	0	0	Unallocated Segment
0	0	1	Incore
0	1	0	Unused
0	1	1	On Disk
1	0	1	Locked Incore
1	1	0	Unused
1	0	0	Unused
1	1	1	Unused

Figure 8-12: Shared Memory State Information

The implied states of Figure 8-12 are as follows:

- **Unallocated Segment**—the segment associated with this segment descriptor has not been allocated for use.
- **Incore**—the shared segment associated with this descriptor has been allocated for use. Therefore, the segment does exist and is currently resident in memory.
- **On Disk**—the shared segment associated with this segment descriptor is currently resident on the swap device.
- **Locked Incore**—the shared segment associated with this segment descriptor is currently locked in memory and will not be a candidate for swapping until the segment is unlocked. Only the super-user may lock and unlock a shared segment.

- **Unused**—this state is currently unused and should never be encountered by the normal user in shared memory handling.

The **shmget(2)** system call is used to perform two tasks when only the **IPC_CREAT** flag is set in the **shmflg** argument that it receives:

- to get a new **shmid** and create an associated shared memory segment data structure for it
- to return an existing **shmid** that already has an associated shared memory segment data structure

The task performed is determined by the value of the **key** argument passed to the **shmget(2)** system call. For the first task, if the **key** is not already in use for an existing **shmid**, a new **shmid** is returned with an associated shared memory segment data structure created for it provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (**IPC_PRIVATE** = 0); when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the **KEY** field for the **shmid** is all zeros.

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a control command (**IPC_EXCL**) can be specified (set) in the **shmflg** argument passed to the system call. The details of using this system call are discussed in the "Using **shmget**" section of this chapter.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Shared Memory" section in this chapter. The creator of the shared memory segment also determines the initial operation permissions for it.

Shared Memory

Once a uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop()**] and control [**shmctl(2)**] can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat(2)** and **shmdt(2)**. Refer to the "Operations for Shared Memory" section in this chapter for details of these system calls.

Shared memory segment control is done by using the **shmctl(2)** system call. It permits you to control the shared memory facility in the following ways:

- to determine the associated data structure status for a shared memory segment (**shmid**)
- to change operation permissions for a shared memory segment
- to remove a particular **shmid** from the UNIX operating system along with its associated shared memory segment data structure
- to lock a shared memory segment in memory
- to unlock a shared memory segment

Refer to the "Controlling Shared Memory" section in this chapter for details of the **shmctl(2)** system call.

Getting Shared Memory Segments

This section gives a detailed description of using the **shmget(2)** system call along with an example program illustrating its use.

Using shmget

The synopsis found in the **shmget(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX operating system. The following line in the synopsis:

```
int shmget (key, size, shmflg)
```

informs you that **shmget(2)** is a function with three formal arguments that returns an integer type value, upon successful completion (**shmid**). The next two lines:

```
key_t key;
int size, shmflg;
```

declare the types of the formal arguments. The variable **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this function upon successful completion is the shared memory identifier (**shmid**) that was discussed earlier.

As declared, the process calling the **shmget(2)** system call must supply three arguments to be passed to the formal **key**, **size**, and **shmflg** arguments.

Shared Memory

A new **shmid** with an associated shared memory data structure is provided if either

- **key** is equal to `IPC_PRIVATE`,

or

- **key** is passed a unique hexadecimal integer, and **shmflg** ANDed with `IPC_CREAT` is `TRUE`.

The value passed to the **shmflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **shmflg** argument. They are collectively referred to as "operation permissions." Figure 8-13 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 8-13: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **shm.h** header file which

can be used for the user (OWNER). They are as follows:

```
SHM_R 0400
SHM_W 0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 8-14 contains the names of the constants that apply to the **shmget()** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 8-14: Control Commands (Flags)

The value for **shmflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
Read by User	=	0 0 4 0 0	0 000 000 100 000 000
shmflg	=	0 1 4 0 0	0 000 001 100 000 000

The **shmflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
msgid = msgget (key, (IPC_CREAT | 0400));
msgid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

457

Shared Memory

As specified by the **shmget(2)** entry in the *System V Reference Manual*, success or failure of this system call depends upon the argument values for **key**, **size**, and **shmflg** or system tunable parameters. The system call will attempt to return a new **shmid** if one of the following conditions is true:

- Key is equal to **IPC_PRIVATE** (0).
- Key does not already have a **shmid** associated with it, and (**shmflg** & **IPC_CREAT**) is "true" (not zero).

The **key** argument can be set to **IPC_PRIVATE** in the following ways:

```
msgid = msgget (IPC_PRIVATE, msgflg);
```

or

```
msgid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the **SHMMNI** system tunable parameter always causes a failure. The **SHMMNI** system tunable parameter determines the maximum number of unique shared memory segments (**shmid**s) in the UNIX operating system.

The second condition is satisfied if the value for **key** is not already associated with a **shmid** and the bitwise ANDing of **shmflg** and **IPC_CREAT** is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the **IPC_CREAT** flag is set (**shmflg** | **IPC_CREAT**). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
msgflg = x 1 x x x  (x = immaterial)
& IPC_CREAT = 0 1 0 0 0

result = 0 1 0 0 0  (not zero)
```

Because the result is not zero, the flag is set or "true." SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **shm**id exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shm**id when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique **shm**id is returned if the system call is successful. Any value for **shmflg** returns a new **shm**id if the **key** equals zero (IPC_PRIVATE).

The system call will fail if the value for the **size** argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment **sizes**.

Refer to the **shmget(2)** manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 8-15) is a menu driven program which allows all possible combinations of using the **shmget(2)** system call to be exercised.

Shared Memory

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-7) by including the required header files as specified by the **shmget(2)** entry in the *System V Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument
- **shmid**—used for returning the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one
- **size**—used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored at the address of the **size** variable (lines 51-54).

The system call is made next, and the result is stored at the address of the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 58). If **shmid** equals -1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the **shmget(2)** system call follows. It is suggested that the source program file be named **shmget.c** and that the executable file be named **shmget**.

Figure 8-15: **shmget(2)** System Call Example

```
1  /*This is a program to illustrate
2  **the shared memory get, shmget(),
3  **system call capabilities.*/

4  #include <sys/types.h>
5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <errno.h>

8  /*Start of main C language program*/
9  main()
10 {
11     key_t key;           /*declare as long integer*/
12     int opperm, flags;
13     int shmid, size, opperm_flags;
14     /*Enter the desired key*/
15     printf("Enter the desired key in hex = ");
16     scanf("%x", &key);
```

(continued on next page)

Shared Memory

```
17      /*Enter the desired octal operation
18      permissions.*/
19      printf("\nEnter the operation\n");
20      printf("permissions in octal = ");
21      scanf("%o", &opperm);

22      /*Set the desired flags.*/
23      printf("\nEnter corresponding number to\n");
24      printf("set the desired flags:\n");
25      printf("No flags           = 0\n");
26      printf("IPC_CREAT              = 1\n");
27      printf("IPC_EXCL                = 2\n");
28      printf("IPC_CREAT and IPC_EXCL    = 3\n");
29      printf("          Flags          = ");
30      /*Get the flag(s) to be set.*/
31      scanf("%d", &flags);

32      /*Check the values.*/
33      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34      key, opperm, flags);

35      /*Incorporate the control fields (flags) with
36      the operation permissions*/
37      switch (flags)
38      {
39      case 0:      /*No flags are to be set.*/
40      /      opperm_flags = (opperm | 0);
41      break;
42      case 1:      /*Set the IPC_CREAT flag.*/
43      opperm_flags = (opperm | IPC_CREAT);
44      break;
45      case 2:      /*Set the IPC_EXCL flag.*/
46      opperm_flags = (opperm | IPC_EXCL);
47      break;
48      case 3:      /*Set the IPC_CREAT and IPC_EXCL flags.*/
49      opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50      }

51      /*Get the size of the segment in bytes.*/
52      printf ("\nEnter the segment");
53      printf ("\nsize in bytes = ");
```

(continued on next page)

```
54     scanf ("%d", &size);

55     /*Call the shmget system call.*/
56     shmid = shmget (key, size, opperm_flags);

57     /*Perform the following if the call is unsuccessful.*/
58     if(shmid == -1)
59     {
60         printf ("\nThe shmget system call failed!\n");
61         printf ("The error number = %d\n", errno);
62     }
63     /*Return the shmud upon successful completion.*/
64     else
65         printf ("\nThe shmud = %d\n", shmud);
66     exit(0);
67 }
```

Shared Memory

This page is intentionally left blank

Controlling Shared Memory

This section gives a detailed description of using the **shmctl(2)** system call along with an example program which allows all of its capabilities to be exercised.

Using shmctl

The synopsis found in the **shmctl(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shm_id, cmd, buf)
int shm_id, cmd;
struct shm_id_ds *buf;
```

The **shmctl(2)** system call requires three arguments to be passed to it, and **shmctl(2)** returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, **shmctl()** returns a -1 .

The **shm_id** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget(2)** system call.

The **cmd** argument can be replaced by one of following control commands (flags):

- **IPC_STAT**—return the status information contained in the associated data structure for the specified **shm_id** and place it in the data structure pointed to by the ***buf** pointer in the user memory area

Controlling Shared Memory

- **IPC_SET**—for the specified **shmid**, set the effective user and group identification, and operation permissions
- **IPC_RMID**—remove the specified **shmid** along with its associated shared memory segment data structure
- **SHM_LOCK**—lock the specified shared memory segment in memory, must be super-user
- **SHM_UNLOCK**—unlock the shared memory segment from memory, must be super-user.

A process must have an effective user identification of **OWNER/CREATOR** or super-user to perform an **IPC_SET** or **IPC_RMID** control command. Only the super-user can perform a **SHM_LOCK** or **SHM_UNLOCK** control command. A process must have read permission to perform the **IPC_STAT** control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-16) is a menu driven program which allows all possible combinations of using the **shmctl(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmctl(2)** entry in the *System V Reference Manual*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer value from the system call
- **shmid**—used to store and pass the shared memory segment identifier to the system call
- **command**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member for the IPC_SET control command that is to be changed
- **shmid_ds**—used to receive the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed
- ***buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

Note that the **shmid_ds** data structure in this program (line 16) uses the data structure located in the **shm.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

Controlling Shared Memory

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **shmids** type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier which is stored at the address of the **shmids** variable (lines 18-20). This is required for every **shmctl(2)** system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the **IPC_STAT** control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).

If the **IPC_SET** control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the choice variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user memory area data structure, and the

system call is made (lines 128-130). Depending upon success or failure, the program returns the same messages as for `IPC_STAT` above.

If the `IPC_RMID` control command (code 3) is selected, the system call is performed (lines 132-135), and the `shmid` along with its associated message queue and data structure are removed from the UNIX operating system. Note that the `*buf` pointer is not required as an argument to perform this control command and its value can be zero or `NULL`. Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the `SHM_LOCK` control command (code 4) is selected, the system call is performed (lines 137,138). Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the `SHM_UNLOCK` control command (code 5) is selected, the system call is performed (lines 140-142). Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the `shmctl(2)` system call follows. It is suggested that the source program file be named `shmctl.c` and that the executable file be named `shmctl`.

Controlling Shared Memory

Figure 8-16: `shmctl(2)` System Call Example

```
1  /*This is a program to illustrate
2  **the shared memory control, shmctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode;
15     int rtrn, shmid, command, choice;
16     struct shm_id_s shm_id_s, *buf;
17     buf = &shm_id_s;

18     /*Get the shmid, and command.*/
19     printf("Enter the shmid = ");
20     scanf("%d", &shmid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");

23     printf("IPC_STAT   = 1\n");
24     printf("IPC_SET    = 2\n");
25     printf("IPC_RMID   = 3\n");
26     printf("SHM_LOCK   = 4\n");
27     printf("SHM_UNLOCK = 5\n");
28     printf("Entry     = ");
29     scanf("%d", &command);

30     /*Check the values.*/
31     printf ("\nshmid =%d, command = %d\n",
32            shmid, command);

33     switch (command)
34     {
35     case 1: /*Use shmctl() to duplicate
```

(continued on next page)

```

36         the data structure for
37         shmId in the shmId_ds area pointed
38         to by buf and then print it out.*/
39     rtn = shmctl(shmid, IPC_STAT,
40         buf);
41     printf ("\nThe USER ID = %d\n",
42         buf->shm_perm.uid);
43     printf ("The GROUP ID = %d\n",
44         buf->shm_perm.gid);
45     printf ("The creator's ID = %d\n",
46         buf->shm_perm.cuid);
47     printf ("The creator's group ID = %d\n",
48         buf->shm_perm.cgid);
49     printf ("The operation permissions = 0%o\n",
50         buf->shm_perm.mode);
51     printf ("The slot usage sequence\n");

52     printf ("number = 0%x\n",
53         buf->shm_perm.seq);
54     printf ("The key= 0%x\n",
55         buf->shm_perm.key);
56     printf ("The segment size = %d\n",
57         buf->shm_segsz);
58     printf ("The pid of last shmop = %d\n",
59         buf->shm_lpid);
60     printf ("The pid of creator = %d\n",
61         buf->shm_cpid);
62     printf ("The current # attached = %d\n",
63         buf->shm_nattch);
64     printf("The in memory # attached = %d\n",
65         buf->shm_cnattach);
66     printf("The last shmat time = %d\n",
67         buf->shm_atime);
68     printf("The last shmdt time = %d\n",
69         buf->shm_dtime);
70     printf("The last change time = %d\n",
71         buf->shm_ctime);
72     break;

        /* Lines 73 - 87 deleted */

88     case 2:    /*Select and change the desired

```

(continued on next page)

Controlling Shared Memory

```

89             member(s) of the data structure.*/

90     /*Get the original data for this shmId
91     data structure first.*/
92     rtrn = shmctl(shmId, IPC_STAT, buf);

93     printf("\nEnter the number for the\n");
94     printf("member to be changed:\n");
95     printf("shm_perm.uid = 1\n");
96     printf("shm_perm.gid = 2\n");
97     printf("shm_perm.mode = 3\n");
98     printf("Entry      = ");
99     scanf("%d", &choice);
100    /*Only one choice is allowed per
101    pass as an illegal entry will
102    cause repetitive failures until
103    shmId_ds is updated with
104    IPC_STAT.*/

105    switch(choice){
106    case 1:
107        printf("\nEnter USER ID = ");
108        scanf ("%d", &uid);
109        buf->shm_perm.uid = uid;
110        printf("\nUSER ID = %d\n",
111        buf->shm_perm.uid);
112        break;

113    case 2:
114        printf("\nEnter GROUP ID = ");
115        scanf("%d", &gid);
116        buf->shm_perm.gid = gid;
117        printf("\nGROUP ID = %d\n",
118        buf->shm_perm.gid);
119        break;

120    case 3:
121        printf("\nEnter MODE = ");
122        scanf("%o", &mode);
123        buf->shm_perm.mode = mode;
124        printf("\nMODE = %o\n",
125        buf->shm_perm.mode);

```

(continued on next page)


```
126         break;
127     }
128     /*Do the change.*/
129     rtn = shmctl(shmid, IPC_SET,
130         buf);
131     break;

132     case 3: /*Remove the shmid along with its
133         associated
134         data structure.*/
135     rtn = shmctl(shmid, IPC_RMID, NULL);
136     break;

137     case 4: /*Lock the shared memory segment*/
138     rtn = shmctl(shmid, SHM_LOCK, NULL);
139     break;
140     case 5: /*Unlock the shared memory
141         segment.*/
142     rtn = shmctl(shmid, SHM_UNLOCK, NULL);
143     break;
144 }
145 /*Perform the following if the call is unsuccessful.*/
146 if(rtn == -1)
147 {
148     printf ("\nThe shmctl system call failed!\n");
149     printf ("The error number = %d\n", errno);
150 }
151 /*Return the shmid upon successful completion.*/
152 else
153     printf ("\nShmctl was successful for shmid = %d\n",
154         shmid);
155     exit (0);
156 }
```

Controlling Shared Memory

This page is intentionally left blank

Operations for Shared Memory

This section gives a detailed description of using the **shmat(2)** and **shmdt(2)** system calls, along with an example program which allows all of their capabilities to be exercised.

Using shmop

The synopsis found in the **shmop(2)** entry in the *System V Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
    int shmid;
    char *shmaddr;
    int shmflg;

int shmdt (shmaddr)
    char *shmaddr;
```

Attaching a Shared Memory Segment

The **shmat(2)** system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. Upon successful completion, this value will be the address in core memory where the process is attached to the shared memory segment and when unsuccessful it will be a `-1`.

The **shmid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget(2)** system call.

Operations for Shared Memory

The **shmaddr** argument can be zero or user supplied when passed to the **shmat(2)** system call. If it is zero, the UNIX operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the UNIX operating system would pick. The following illustrates some typical address ranges; these are for the SUPERMAX Computer:

0x400000
0x500000
0x600000
0x700000

Note that these addresses are in chunks of 10,000 hexadecimal. It would be wise to let the operating system pick addresses so as to improve portability.

The **shmflg** argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat()** system call.

Further details are discussed in the example program for **shmop()**. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Detaching Shared Memory Segments

The **shmdt(2)** system call requires one argument to be passed to it, and **shmdt(2)** returns an integer value.

Upon successful completion, zero is returned; and when unsuccessful, **shmdt(2)** returns a -1.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-17) is a menu driven program which allows all possible combinations of using the **shmat(2)** and **shmdt(2)** system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop(2)** entry in the *System V Reference Manual*. Note that in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **flags**—used to store the codes of SHM_RND or SHM_RDONLY for the **shmat(2)** system call
- **addr**—used to store the address of the shared memory segment for the **shmat(2)** and **shmdt(2)** system calls
- **i**—used as a loop counter for attaching and detaching
- **attach**—used to store the desired number of attach operations
- **shmid**—used to store and pass the desired shared memory segment identifier
- **shmflg**—used to pass the value of flags to the **shmat(2)** system call
- **retrn**—used to store the return values from both system calls

Operations for Shared Memory

- **detach**—used to store the desired number of detach operations

This example program combines both the **shmat(2)** and **shmdt(2)** system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the attach variable (lines 17-21).

A loop is entered using the attach variable and the *i* counter (lines 23-70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored at the address of the **shmid** variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored at the address of the **addr** variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored at the address of the flags variable (line 45). The flags variable is tested to determine the code to be stored for the **shmflg** variable used to pass them to the **shmat(2)** system call (lines 46-57). The system call is made (line 60). If successful, a message stating so is displayed along with the attach address (lines 66-68). If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the detach variable (line 76).

A loop is entered using the detach variable and the i counter (lines 78-95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the **addr** variable (line 84). Then, the **shmdt(2)** system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the **shmop(2)** system calls follows. It is suggested that the program be put into a source file called **shmop.c** and then into an executable file called **shmop**.

Figure 8-17: **shmop()** System Call Example

```

1  /*This is a program to illustrate
2  **the shared memory operations, shmop(),
3  **system call capabilities.
4  */
5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int flags, addr, i, attach;
15     int shmid, shmflg, retn, detach;
16     /*Loop for attachments by this process.*/
17     printf("Enter the number of\n");
18     printf("attachments for this\n");
19     printf("process (1-4).\n");
20     printf("    Attachments = ");
21     scanf("%d", &attach);
22     printf("Number of attaches = %d\n", attach);
23     for(i = 1; i <= attach; i++) {

```

(continued on next page)

Operations for Shared Memory

```

24      /*Enter the shared memory ID.*/
25      printf("\nEnter the shmid of\n");
26      printf("the shared memory segment to\n");
27      printf("be operated on = ");
28      scanf("%d", &shmid);
29      printf("\nshmid = %d\n", shmid);
30      /*Enter the value for shmaddr.*/
31      printf("\nEnter the value for\n");
32      printf("the shared memory address\n");
33      printf("in hexadecimal:\n");
34      printf("      Shmaddr = ");
35      scanf("%x", &addr);
36      printf("The desired address = 0x%x\n", addr);

37      /*Specify the desired flags.*/
38      printf("\nEnter the corresponding\n");
39      printf("number for the desired\n");
40      printf("flags:\n");
41      printf("SHM_RND                = 1\n");
42      printf("SHM_RDONLY                = 2\n");
43      printf("SHM_RND and SHM_RDONLY = 3\n");
44      printf("      Flags                = ");
45      scanf("%d", &flags);

46      switch(flags)
47      {
48      case 1:
49          shmflg = SHM_RND;
50          break;
51      case 2:
52          shmflg = SHM_RDONLY;
53          break;
54      case 3:
55          shmflg = SHM_RND | SHM_RDONLY;
56          break;
57      }
58      printf("\nFlags = %o\n", shmflg);
59      /*Do the shmat system call.*/
60      retrn = (int)shmat(shmid, addr, shmflg);
61      if(retrn == -1) {
62          printf("\nShmat failed. ");
63          printf("Error = %d\n", errno);

```

(continued on next page)


```
64     }
65     else {
66         printf ("\nShmat was successful\n");
67         printf("for shmid = %d\n", shmid);
68         printf("The address = 0x%x\n", retrn);
69     }
70 }

71 /*Loop for detachments by this process.*/
72 printf("Enter the number of\n");
73 printf("detachments for this\n");
74 printf("process (1-4).\n");
75 printf("      Detachments = ");
76 scanf("%d", &detach);
77 printf("Number of attaches = %d\n", detach);
78 for(i = 1; i <= detach; i++) {
79     /*Enter the value for shmaddr.*/
80     printf("\nEnter the value for\n");
81     printf("the shared memory address\n");
82     printf("in hexadecimal:\n");
83     printf("      Shmaddr = ");
84     scanf("%x", &addr);
85     printf("The desired address = 0x%x\n", addr);

86     /*Do the shmdt system call.*/
87     retrn = (int)shmdt(addr);
88     if(retrn == -1) {
89         printf("Error = %d\n", errno);
90     }
91     else {
92         printf ("\nShmdt was successful\n");
93         printf("for address = 0%x\n", addr);
94     }
95 }
96 }
```

Operations for Shared Memory

This page is intentionally left blank

Chapter 9: curses/terminfo

	Page
Introduction.....	9- 1
Overview	9- 3
What is curses ?.....	9- 3
What is terminfo ?.....	9- 5
How curses and terminfo Work Together	9- 6
Other Components of the Terminal Information Utilities .	9- 7
Working with curses Routines.....	9- 9
What Every curses Program Needs	9-10
The Header File <curses.h>	9-10
The Routines initscr() , refresh() , endwin()	9-11
Compiling a curses Program.....	9-13
Running a curses Program	9-13
More about initscr() and Lines and Columns.....	9-14
More about refresh() and Windows	9-14
Getting Simple Output and Input.....	9-20
Output	9-20
Input.....	9-33
Controlling Output and Input.....	9-41
Output Attributes.....	9-41
Bells, Whistles, and Flashing Lights	9-46
Input Options.....	9-47
Building Windows and Pads	9-53
Output and Input	9-53
The Routines wnoutrefresh() and doupdate()	9-54

Table of Contents

	Page
New Windows.....	9-59
Using Advanced curses Features.....	9-63
Routines for Drawing Lines and Other Graphics.....	9-63
Routines for Using Soft Labels.....	9-65
Working with More than One Terminal.....	9-67
Working with terminfo Routines.....	9-69
What Every terminfo Program Needs.....	9-70
Compiling and Running a terminfo Program.....	9-71
An Example terminfo Program.....	9-71
Working with the terminfo Database.....	9-77
Writing Terminal Descriptions.....	9-77
Name the Terminal.....	9-78
Learn About the Capabilities.....	9-79
Specify Capabilities.....	9-80
Basic Capabilities.....	9-83
Screen-Oriented Capabilities.....	9-83
Keyboard-entered Capabilities.....	9-84
Parameter String Capabilities.....	9-85
Compile the Description.....	9-87
Test the Description.....	9-88
Comparing or Printing terminfo Descriptions.....	9-89
Converting a termcap Description to a terminfo Description.....	9-90
curses Program Examples.....	9-91
The editor Program.....	9-91
The highlight Program.....	9-99
The scatter Program.....	9-101

Table of Contents

	Page
The show Program.....	9 – 103
The two Program.....	9 – 105
The window Program.....	9 – 109

485

Table of Contents

This page is intentionally left blank

Introduction

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the Terminal Information Utilities package, commonly called **curses/terminfo**, to write screen management programs on a UNIX system. This package includes a library of C routines, a database, and a set of UNIX system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to **curses(3X)** and **terminfo(4)** in the *System V Reference Manual* for more information. Keep the manual close at hand; you'll find it invaluable when you want to know more about one of these routines or about other routines not discussed here.

Because the routines are compiled C functions, you should be familiar with the C programming language before using **curses/terminfo**. You should also be familiar with the UNIX system/C language standard I/O package (see "System Calls and Subroutines" and "Input/Output" in Chapter 2 and **stdio(3S)**). With that knowledge and an appreciation for the UNIX philosophy of building on the work of others, you can design screen management programs for many purposes.

This chapter has five sections:

- Overview

This section briefly describes **curses**, **terminfo**, and the other components of the Terminal Information Utilities package.

Introduction

- Working with **curses** Routines

This section describes the basic routines making up the **curses(3X)** library. It covers the routines for writing to a screen, reading from a screen, and building windows. It also covers routines for more advanced screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time. Many examples are included to show the effect of using these routines.

- Working with **terminfo** Routines

This section describes the routines in the **curses** library that deal directly with the **terminfo** database to handle certain terminal capabilities, such as programming function keys.

- Working with the **terminfo** Database

This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

- **curses** Program Examples

This section includes six programs that illustrate uses of **curses** routines.

Overview

What is curses?

curses(3X) is the library of routines that you use to write screen management programs on the UNIX system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw()** that behaves much like **printf(3S)** and another routine **getch()** that behaves like **getc(3S)**. The automatic teller program at your bank might use **printw()** to print its menus and **getch()** to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX system screen editor **vi(1)** might also use these and other **curses** routines.

The **curses** routines are usually located in **/usr/lib?/libcurses.a**, (where **?** is the **TARGETMC**). To compile a program using these routines, you must use the **cc(1)** command and include **-lcurses** on the command line so that the link editor can locate and load them:

```
cc file.c -lcurses -o file
```

The name **curses** comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with **curses** routines and edited the sentence

```
curses/terminfo is a great package for creating screens.  
to read
```

```
curses/terminfo is the best package for creating screens.  
the program would output only the best in place of a great. The  
other characters would be preserved. Because the amount of data  
transmitted—the output—is minimized, cursor optimization is also  
referred to as output optimization.
```

Overview

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which a **curses** program is run. This means that the **curses** library can do whatever is required to update many different terminal types. It searches the **terminfo** database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your UNIX system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple **curses** program. It uses some of the basic **curses** routines to move a cursor to the middle of a terminal screen and print the character string **BullsEye**. Each of these routines is described in the following section "Working with **curses** Routines" in this chapter. For now, just look at their names and you will get an idea of what each of them does:

```
#include <curses.h>

main()
{
    initscr();

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();
    addstr("Eye");
    refresh();
    endwin();
}
```

Figure 9-1: A Simple **curses** Program

What is terminfo?

terminfo refers to both of the following:

- It is a group of routines within the **curses** library that handles certain terminal capabilities. You can use these routines to program function keys, if your terminal has programmable keys, or write filters, for example. Shell programmers, as well as C programmers, can use the **terminfo** routines in their programs.
- It is a database containing the descriptions of many terminals that can be used with **curses** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

Each terminal description in the database is a separate, compiled file. You use the source code that **terminfo(4)** describes to create these files and the command **tic(1M)** to compile them.

The compiled files are normally located in the directories **/usr/lib/terminfo/?**. These directories have single character names, each of which is the first character in the name of a terminal. For example, the standard entry for a terminal on the SUPERMAX computer is located in the file **/usr/lib/terminfo/T/T3-24-C80**.

Here's a simple shell script that uses the **terminfo** database.

Overview

```
# Clear the screen and show the 0,0 position.
#
tput clear
tput cup 0 0      # or tput home
echo "<- this is 0 0"

#
# Show the 5,10 position.
#
tput cup 5 10
echo "<- this is 5 10"
```

Figure 9-2: A Shell Script Using **terminfo** Routines

How **curses** and **terminfo** Work Together

A screen management program with **curses** routines refers to the **terminfo** database at run time to obtain the information it needs about the terminal being used—what we'll call the current terminal from here on.

For example, suppose you are using a terminal with 24 lines and 80 columns. To run the simple **curses** program shown in Figure 9-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the BullsEye in the middle of it. The description of the terminal **terminfo** database has this information. All the **curses** program needs to know before it goes looking for the information is the name of your terminal. You tell the program the name by putting it in the environment variable **\$TERM** when you log in or by setting and exporting **\$TERM** in your **.profile** file (see **profile(4)**). Knowing **\$TERM**, a **curses** program run on the current terminal can search the **terminfo** database to find the correct terminal description.

For example, assume that the following example lines are in a **.profile**:

```
TERM=T3-24-C80
export TERM
```

The first line names the terminal type, and the second line exports it. (See **profile(4)** in the *System V Reference Manual*.) If you had these lines in your **.profile** and you ran a **curses** program, the program would get the information that it needs about your terminal from the file `/usr/lib/terminfo/T/T3-24-C80`, which provides a match for **\$TERM**.

Other Components of the Terminal Information Utilities

We said earlier that the Terminal Information Utilities is commonly referred to as **curses/terminfo**. The package, however, has other components. We've mentioned some of them, for instance **tic(1M)**. Here's a complete list of the components discussed in this tutorial:

- | | |
|----------------------|--|
| captainfo(1M) | a tool for converting terminal descriptions developed on earlier releases of the UNIX system to terminfo descriptions |
| curses(3X) | |
| infocmp(1M) | a tool for printing and comparing compiled terminal descriptions |
| tabs(1) | a tool for setting non-standard tab stops |
| terminfo(4) | |
| tic(1M) | a tool for compiling terminal descriptions for the terminfo database |

Overview

tput(1) a tool for initializing the tab stops on a terminal and for outputting the value of a terminal capability

We also refer to **profile(4)**, **scr_dump(4)**, **term(4)**, and **terminology(1)**. For more information about any of these components, see the *System V Reference Manual* and the *Virtual Terminal Interface Guide*.

Working with **curses** Routines

This section describes the basic **curses** routines for creating interactive screen management programs. It begins by describing the routines and other program components that every **curses** program needs to work properly. Then it tells you how to compile and run a **curses** program. Finally, it describes the most frequently used **curses** routines that

- write output to and read input from a terminal screen
- control the data output and input – for example, to print output in bold type or prevent it from echoing (printing back on a screen)
- manipulate multiple screen images (windows)
- draw simple graphics
- manipulate soft labels on a terminal screen
- send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, we include simple example programs as the routines are introduced. We also refer to a group of larger examples located in the section "**curses** Program Examples" in this chapter. These larger examples are more challenging; they sometimes make use of routines not discussed here. Keep the **curses(3X)** manual page handy.

What Every curses Program Needs

All **curses** programs need to include the header file `< curses.h >` and call the routines **initscr()**, **refresh()** or similar related routines, and **endwin()**.

The Header File `< curses.h >`

The header file `< curses.h >` defines several global variables and data structures and defines several **curses** routines as macros.

To begin, let's consider the variables and data structures defined. `< curses.h >` defines all the parameters used by **curses** routines. It also defines the integer variables **LINES** and **COLS**; when a **curses** program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine **initscr()** described below. The header file defines the constants **OK** and **ERR**, too. Most **curses** routines have return values; the **OK** value is returned if a routine is properly completed, and the **ERR** value if some error occurs.

NOTE

LINES and **COLS** are external (global) variables that represent the size of a terminal screen. Two similar variables, **\$LINES** and **\$COLUMNS**, may be set in a user's shell environment; a **curses** program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this chapter, we will use the **\$** to distinguish them from the C declarations in the `< curses.h >` header file.

For more information about these variables, see the following sections "The Routines **initscr()**, **refresh()**, and **endwin()**" and "More about **initscr()** and Lines and Columns."

Now let's consider the macro definitions. `< curses.h >` defines many **curses** routines as macros that call other macros or **curses** routines. For instance, the simple routine **refresh()** is a macro. The line

```
#define refresh() wrefresh(stdscr)
```

shows when **refresh** is called, it is expanded to call the **curses**

routine **wrefresh()**. The latter routine in turn calls the two **curses** routines **wnoutrefresh()** and **doupdate()**. Many other routines also group two or three routines together to achieve a particular result.



Macro expansion in **curses** programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about **<curses.h>**: it automatically includes **<stdio.h>** and the **<termio.h>** tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines **initscr()**, **refresh()**, and **endwin()**

The routines **initscr()**, **refresh()**, and **endwin()** initialize a terminal screen to an "in **curses** state," update the contents of the screen, and restore the terminal to an "out of **curses** state," respectively. Use the simple program that we introduced earlier to learn about each of these routines:

```
#include <curses.h>
main()
{
    initscr();      /* initialize terminal settings and <curses.h>
                   data structures and variables */

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();     /* send output to (update) terminal screen */
    addstr("Eye");
    refresh();     /* send more output to terminal screen */
    endwin();      /* restore all terminal settings */
}
```

Figure 9-3: The Purposes of **initscr()**, **refresh()**, and **endwin()** in a Program

Working with curses Routines

A **curses** program usually starts by calling **initscr()**; the program should call **initscr()** only once. Using the environment variable **\$TERM** as the section "How **curses** and **terminfo** Work Together" describes, this routine determines what terminal is being used. It then initializes all the declared data structures and other variables from **<curses.h>**. For example, **initscr()** would initialize **LINES** and **COLS** for the sample program on whatever terminal it was run. If **\$TERM=T3-24-C80** were used, this routine would initialize **LINES** to 24 and **COLS** to 80. Finally, this routine writes error messages to **stderr** and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move()** and **addstr()** in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. Then the line

```
addstr("Bulls");
```

says to write the character string **Bulls**. For example, if **\$TERM=T3-24-C805** were used, these routines would position the cursor and write the character string at (11,36).

NOTE

All **curses** routines that move the cursor move it from its home position in the upper left corner of a screen. The **(LINES, COLS)** coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common 'x,y' order of screen (or graph) coordinates. The **-1** in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like **move()** and **addstr()** do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh()** is called. Before this, an internal representation of the screen called a window is updated. This is a very important concept, which we discuss below under "More about **refresh()** and Windows."

Finally, a **curses** program ends by calling **endwin()**. This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling a curses Program

You compile programs that include **curses** routines as C language programs using the **cc(1)** command (documented in the *System Reference Manual*), which invokes the C compiler (see Chapter 2 in this guide for details).

The routines are usually stored in the library **/usr/lib?/libcurses.a**, (where ? is TARGETMC). To direct the link editor to search this library, you must use the **-l** option with the **cc** command.

The general command line for compiling a **curses** program follows:

```
cc file.c -lcurses -o file
```

file.c is the name of the source program; and *file* is the executable object module.

Running a curses Program

curses programs count on certain information being in a user's environment to run properly. Specifically, users of a **curses** program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
```

For an explanation of these lines, see the section "How **curses** and **terminfo** Work Together" in this chapter. Users of a **curses** program could also define the environment variables **\$LINES**, **\$COLUMNS**, and **\$TERMINFO** in their **.profile** files. However, unlike **\$TERM**, these variables do not have to be defined.

Working with curses Routines

If a **curses** program does not run as expected, you might want to debug it with **sdb**(1), which is documented in the *System V Reference Manual*). When using **sdb**, you have to keep a few points in mind. First, a **curses** program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **sdb**, however, may cause changes to the contents of the screen of which the **curses** program is not aware.

Second, a **curses** program outputs to a window until **refresh()** or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on **curses** routines that are macros, such as **refresh()**, does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh()** instead of **refresh()**. See the above section, "The Header File `<curses.h>`," for more information about macros.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, **initscr()** sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment **\$LINES** and **\$COLUMNS**.

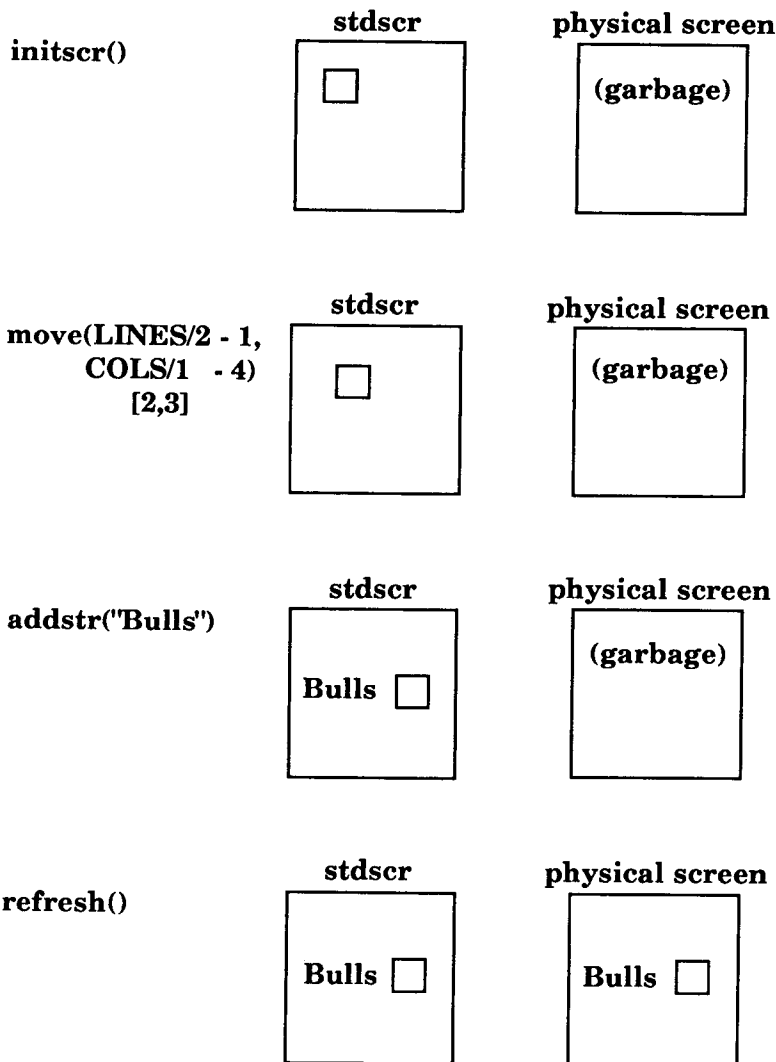
More about refresh() and Windows

As mentioned above, **curses** routines do not update a terminal until **refresh()** is called. Instead, they write to an internal representation of the screen called a window. When **refresh()** is called, all the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like a buffer does when you use a UNIX system editor. When you invoke **vi(1)**, for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of **curses** routines, they change the contents of a window. The changes become part of the current terminal screen only when **refresh()** is called.

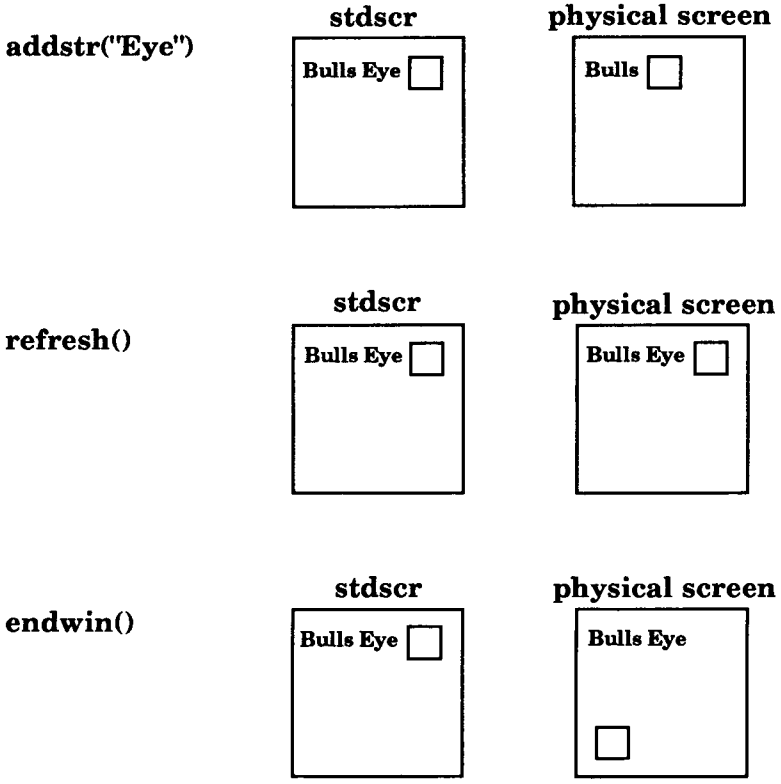
<curses.h> supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using **curses** routines. The header file defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh()** is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like **stdscr**. A **curses** program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as is possible. Figure 9-4 illustrates what happens when you execute the sample curses program that prints **BullsEye** at the center of a terminal screen (see Figure 9-1). Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh()** is called. This **refresh()** clears the screen and updates it with the current contents of **stdscr**.

Working with curses Routines



502

Figure 9-4: The Relationship between **stdscr** and a Terminal Screen
(Sheet 1 of 2)



503

Figure 9-4: The Relationship Between **stdscr** and a Terminal Screen
(Sheet 2 of 2)

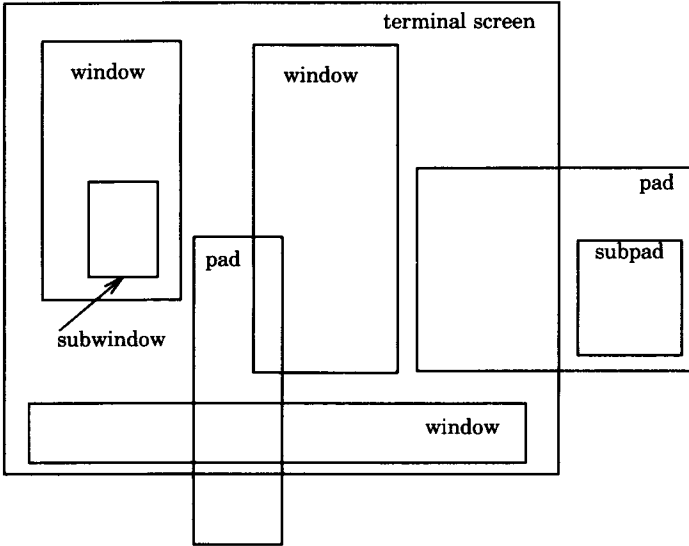
Working with curses Routines

You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It's possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It's also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some **curses** routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 9-5 represents what a pad, a subwindow, and some other windows could look like in comparison to a terminal screen.



505

Figure 9-5: Multiple Windows and Pads Mapped to a Terminal Screen

NOTE

The section "Building Windows and Pads" in this chapter describes the routines you use to create and use them. If you'd like to see a **curses** program with windows now, you can turn to the **window** program under the section "curses Program Examples" in this chapter.

Getting Simple Output and Input

Output

The routines that **curses** provides for writing to **stdscr** are similar to those provided by the **stdio(3S)** library for writing to a file. They let you

- write a character at a time – **addch()**
- write a string – **addstr()**
- format a string from a variety of input arguments – **printw()**
- move a cursor or move a cursor and print character(s) – **move()**, **mvaddch()**, **mvaddstr()**, **mvprintw()**
- clear a screen or a part of it – **clear()**, **erase()**, **clrtoeol()**, **clrtoobot()**

Following are descriptions and examples of these routines.



The **curses** library provides its own set of output and input functions. You should not use other I/O routines or system calls, like **read(2)** and **write(2)**, in a **curses** program. They may cause undesirable results when you run the program.

NAME

addch()

SYNOPSIS

```
#include <curses.h>
```

```
int addch(ch)
cetype ch;
```

NOTES

- **addch()** writes a single character to **stdscr**.
- The character is of the type **cetype**, which is defined in **<curses.h>**. **cetype** contains data and attributes (see "Output Attributes" in this chapter for information about attributes).
- When working with variables of this type, make sure you declare them as **cetype** and not as the basic type (for example, **short**) that **cetype** is declared to be in **<curses.h>**. This will ensure future compatibility.
- **addch()** does some translations. For example, it converts
 - the **<NL>** character to a clear to end of line and a move to the next line
 - the tab character to an appropriate number of blanks
 - other control characters to their **^X** notation
- **addch()** normally returns **OK**. The only time **addch()** returns **ERR** is after adding a character to the lower right-hand corner of a window that does not scroll.
- **addch()** is a macro.

Getting Simple Output and Input

EXAMPLE

```
#include <curses.h>
```

```
main()
{
    initscr();
    addch('a');
    refresh();
    endwin();
}
```

The output from this program will appear as follows:



```
a
```

```
$□
```

Also see the **show** program under "curses Example Programs" in this chapter.

NAME

addstr()

SYNOPSIS

#include <curses.h>

int addstr(str)

char * str;

NOTES

- **addstr()** writes a string of characters to **stdscr**.
- **addstr()** calls **addch()** to write each character.
- **addstr()** follows the same translation rules as **addch()**.
- **addstr()** returns **OK** on success and **ERR** on error.
- **addstr()** is a macro.

EXAMPLE

Recall the sample program that prints the character string **BullsEye**. See Figures 9-1, 9-2, and 9-4.

Getting Simple Output and Input

NAME

printw()

SYNOPSIS

#include <curses.h>

int printw(fmt [,arg...])

char * fmt

NOTES

- **printw()** handles formatted printing on **stdscr**.
- Like **printf**, **printw()** takes a format string and a variable number of arguments.
- Like **addstr()**, **printw()** calls **addch()** to write the string.
- **printw()** returns **OK** on success and **ERR** on error.

EXAMPLE

```
#include <curses.h>

main()
{
    char* title = "Not specified";
    int no = 0;

    /* Missing code. */

    initscr();

    /* Missing code. */

    printw("%s is not in stock.\n", title);
    printw("Please ask the cashier to order %d\
          for you.\n", no);

    refresh();
    endwin();
}
```

The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.
```

```
$□
```

Getting Simple Output and Input

NAME

move()

SYNOPSIS

```
#include < curses.h >
```

```
int move(y, x);
```

```
int y, x;
```

NOTES

- **move()** positions the cursor for **stdscr** at the given row **y** and the given column **x**.
- Notice that **move()** takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section "The Routines **initscr()**, **refresh()**, and **endwin()**" for more information.
- **move()** may be combined with the write functions to form
 - **mvaddch(y, x, ch)**, which moves to a given position and prints a character
 - **mvaddstr(y, x, str)**, which moves to a given position and prints a string of characters
 - **mvprintw(y, x, fmt [,arg...])**, which moves to a given position and prints a formatted string.
- **move()** returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (**LINES** - 1, **COLS** - 1) causes an error.
- **move()** is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();      /* Gets <CR>; discussed below. */
    endwin();
}
```

Here's the output generated by running this program:

```
Cursor should be here --> if move() works.
```

```
Press <CR> to end test.
```

After you press **<CR>**, the screen looks like this:

```
Cursor should be here -->
```

```
Press <CR> to end test.
```

```
$ 
```

See the **scatter** program under "curses Program Examples" in this chapter for another example of using **move()**.

Getting Simple Output and Input

NAME

clear() and **erase()**

SYNOPSIS

```
#include <curses.h>
```

```
int clear()
```

```
int erase()
```

NOTES

- Both routines change **stdscr** to all blanks.
- **clear()** also assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase()** and then **clearok()** which clears the physical screen completely on the next call to **refresh()** for **stdscr**. See the **curses(3X)** manual page for more information about **clearok()**.
- **initscr()** automatically calls **clear()**.
- **clear()** always returns **OK**; **erase()** returns no useful value.
- Both routines are macros.

NAME

clrtoeol() and **clrrobot()**

SYNOPSIS

#include <curses.h>

int clrtoeol()

int clrrobot()

NOTES

- **clrtoeol()** changes the remainder of a line to all blanks.
- **clrrobot()** changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

Getting Simple Output and Input

EXAMPLE

The following sample program uses `clrtoobot()`.

```
#include <curses.h>

main()
{
    initscr();
    noecho();
    addstr("Press <CR> to delete from here to the end\
        of the line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrtoobot();
    refresh();
    endwin();
}
```

Here's the output generated by running this program:

```
Press <CR> to delete from here to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to `refresh()`: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press `<CR>`:

Press <CR> to delete from here

\$□

See the **show** and **two** programs under "curses Example Programs" for examples of uses for **clrtoeol()**.

Getting Simple Output and Input

This page is intentionally left blank

Input

curses routines for reading from the current terminal are similar to those provided by the **stdio(3S)** library for reading from a file. They let you

- read a character at a time – **getch()**
- read a **<NL>**-terminated string – **getstr()**
- parse input, converting and assigning selected data to an argument list – **scanw()**

The primary routine is **getch()**, which processes a single input character and then returns that character. This routine is like the C library routine **getchar()(3S)** except that it makes several terminal- or system-dependent options available that are not possible with **getchar()**. For example, you can use **getch()** with the **curses** routine **keypad()**, which allows a **curses** program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch()** and **keypad()** on the **curses(3X)** manual page for more information about **keypad()**.

The following pages describe and give examples of the basic routines for getting input in a screen program.

Input

NAME

getch()

SYNOPSIS

#include <curses.h>

int getch()

NOTES

- **getch()** reads a single character from the current terminal.
- **getch()** returns the value of the character or **ERR** on 'end of file,' receipt of signals, or non-blocking read with no input.
- **getch()** is a macro.
- See the discussions about **echo()**, **noecho()**, **cbreak()**, **nocbreak()**, **raw()**, **noraw()**, **halfdelay()**, **nodelay()**, and **keypad()** below and in **curses(3X)**.

EXAMPLE

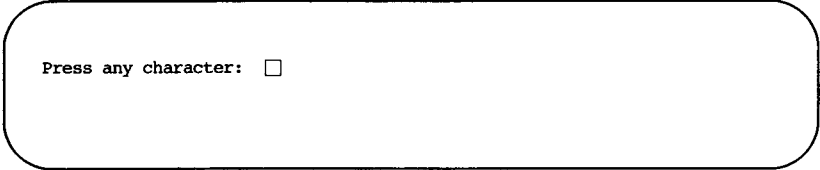
```
#include <curses.h>

main()
{
    int ch;

    initscr();
    cbreak();      /* Explained later in the section */
                  /* "Input Options" */
    addstr("Press any character: ");
    refresh();
    ch = getch();
    printw("The character entered was a '%c'.", ch);
    refresh();
    endwin();
}
```

521

The output from this program follows. The first **refresh()** sends the **addstr()** character string from **stdscr** to the terminal:



```
Press any character: 
```

Then assume that a **w** is typed at the keyboard. **getch()** accepts the character and assigns it to **ch**. Finally, the second **refresh()** is called and the screen appears as follows:

Input

Press any character: w

The character entered was a 'w'.

\$□

For another example of **getch()**, see the **show** program under "curses Example Programs" in this chapter.

NAME

getstr()

SYNOPSIS

#include <curses.h>

int getstr(str)

char * str;

NOTES

- **getstr()** reads characters and stores them in a buffer until a **<CR>**, **<NL>**, or **<ENTER>** is received from **stdscr**. **getstr()** does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr()** is a macro; it calls **getch()** to read each character.
- **getstr()** returns **ERR** if **getch()** returns **ERR** to it. Otherwise it returns **OK**.
- See the discussions about **echo()**, **noecho()**, **cbreak()**, **nocbreak()**, **raw()**, **noraw()**, **halfdelay()**, **nodelay()**, and **keypad()** below and in **curses(3X)**.

Input

EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

    initscr();
    cbreak();          /* Explained later in the section */
                      /* "Input Options" */
    addstr("Enter a character string terminated\
          by <CR>:\n\n");
    refresh();
    getstr(str);
    printw("The string entered was '%s'", str);
    refresh();
    endwin();
}
```

Assume you entered the string 'I enjoy learning about the UNIX system.' The final screen (after entering <CR>) would appear as follows:

```
Enter a character string terminated by <CR>:
```

```
I enjoy learning about the UNIX system.
```

```
The string entered was
```

```
'I enjoy learning about the UNIX system.'
```

```
$□
```

NAME

scanw()

SYNOPSIS

#include <curses.h>

int scanw(fmt [, arg...])

char * fmt;

NOTES

- **scanw()** calls **getstr()** and parses an input line.
- Like **scanf(3S)**, **scanw()** uses a format string to convert and assign to a variable number of arguments.
- **scanw()** returns the same values as **scanf()**.
- See **scanf(3S)** for more information.

Input

EXAMPLE

```
#include <curses.h>

main()
{
    char string[100];
    float number;

    initscr();
    cbreak();           /* Explained later in the */
    echo();             /* section "Input Options" */
    addstr("Enter a number and a string separated by\
          a comma: ");

    refresh();
    scanw("%f,%s",&number,string);
    clear();
    printw("The string was \"%s\"and the number
          was %f.",string,number);

    refresh();
    endwin();
}
```

Notice the two calls to **refresh()**. The first call updates the screen with the character string passed to **addstr()**, the second with the string returned from **scanw()**. Also notice the call to **clear()**. Assume you entered the following when prompted: **2,twin**. After running this program, your terminal screen would appear, as follows:

```
The string was "twin" and the number was 2.000000.
```

```
$□
```

Controlling Output and Input

Output Attributes

When we talked about **addch()**, we said that it writes a single character of the type **chtype** to **stdscr**. **chtype** has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, bold, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine **attrset()** and related **curses** routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- **A_BLINK** blinking
- **A_BOLD** extra bright or bold
- **A_DIM** half bright
- **A_REVERSE** reverse video
- **A_STANDOUT** a terminal's best highlighting mode
- **A_UNDERLINE** underlining
- **A_ALTCHARSET** alternate character set (see the section "Drawing Lines and Other Graphics" in this chapter).

To use these attributes, you must pass them as arguments to **attrset()** and related routines; they can also be ORed with the bitwise OR (**|**) to **addch()**.

Controlling Output and Input

NOTE

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a **curses** program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:

```
...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Attributes can be turned on singly, such as **attrset(A_BOLD)** in the example, or in combination. To turn on blinking bold text, for example, you would use **attrset(A_BLINK | A_BOLD)**. Individual attributes can be turned on and off with the **curses** routines **attron()** and **attroff()** without affecting other attributes. **attrset(0)** turns all attributes off.

Notice the attribute called **A_STANDOUT**. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the **A_STANDOUT** attribute is recommended. Two convenient functions, **standout()** and **standend()** can be used to turn on and off this attribute. **standend()**, in fact, turns off all attributes.

In addition to the attributes listed above, there are two bit masks called `A_CHARTEXT` and `A_ATTRIBUTES`. You can use these bit masks with the `curses` function `inch()` and the C logical AND (`&`) operator to extract the character or attributes of a position on a terminal screen. See the discussion of `inch()` on the `curses(3X)` manual page.

Following are descriptions of `attrset()` and the other `curses` routines that you can use to manipulate attributes.

Controlling Output and Input

NAME

attron(), **attrset()**, and **attroff()**

SYNOPSIS

```
#include < curses.h >
```

```
int attron( attrs )  
chtype attrs;
```

```
int attrset( attrs )  
chtype attrs;
```

```
int attroff( attrs )  
chtype attrs;
```

NOTES

- **attron()** turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in **<curses.h>**.
- **attrset()** turns on the requested attributes **attrs** instead of any that are currently turned on.
- **attroff()** turns off the requested attributes **attrs** if they are on.
- The attributes may be combined using the bitwise OR (|).
- All return **OK**.

EXAMPLE

See the **highlight** program under "**curses** Example Programs" in this chapter.

NAME

standout() and **standend()**

SYNOPSIS

#include <curses.h>

int standout()

int standend()

NOTES

- **standout()** turns on the preferred highlighting attribute, **A_STANDOUT**, for the current terminal. This routine is equivalent to **attron(A_STANDOUT)**.
- **standend()** turns off all attributes. This routine is equivalent to **attrset(0)**.
- Both always return **OK**.

EXAMPLE

See the **highlight** program under "**curses** Example Programs" in this chapter.

Bells, Whistles, and Flashing Lights

Occasionally, you may want to get a user's attention. Two **curses** routines were designed to help you do this. They let you ring the terminal's chimes and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine **beep()** can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep()** will flash the screen.)

NAME

beep() and **flash()**

SYNOPSIS

```
#include <curses.h>
```

```
int flash()
```

```
int beep()
```

NOTES

- **flash()** tries to flash the terminal's screen, if possible, and, if not, tries to ring the terminal bell.
- **beep()** tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- Neither returns any useful value.

Input Options

The UNIX system does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed
- interprets line editing keys.
- interprets a CTRL-D (control d) as end of file (EOF)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates <CR> to <NL>

Because a **curses** program maintains total control over the screen, **curses** turns off echoing on the UNIX system and does echoing itself. At times, you may not want the UNIX system to process other characters in the standard way in an interactive screen management program. Some **curses** routines, **noecho()** and **cbreak()**, for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Figure 9-6 shows some of the major routines for controlling input.

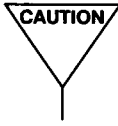
Every **curses** program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak()**, **raw()**, **nocbreak()**, or **noraw()** mode. Although the **curses** program starts up in **echo()** mode, as Figure 9-6 shows, none of the other modes are guaranteed.

The combination of **noecho()** and **cbreak()** is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The **curses** routine **noecho()** is designed for this purpose. However, when **noecho()** turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak()** causes these characters to be uninterpreted.

Controlling Output and Input

Input Options	Characters	
	Interpreted	Uninterpreted
Normal 'out of courses state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal courses 'start up state'	echoing (simulated)	All else undefined.
cbreak() and echo()	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and noecho()	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and noecho()	break, quit stripping erase, kill EOF	echoing
nocbreak() and echo()	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of cbreak())		break, quit stripping

Figure 9-6: Input Option Settings for **courses** Programs



Do not use the combination **nocbreak()** and **noecho()**. If you use it in a program and also use **getch()**, the program will go in and out of **cbreak()** mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Figure 9-6, you can use the **curses** routines **noraw()**, **halfdelay()**, and **nodelay()** to control input. See the **curses(3X)** manual page for discussions of these routines.

The next few pages describe **noecho()**, **cbreak()** and the related routines **echo()** and **nocbreak()** in more detail.

Controlling Output and Input

NAME

echo() and **noecho()**

SYNOPSIS

```
#include <curses.h>
```

```
int echo()
```

```
int noecho()
```

NOTES

- **echo()** turns on echoing of characters by **curses** as they are read in. This is the initial setting.
- **noecho()** turns off the echoing.
- Neither returns any useful value.
- **curses** programs may not run properly if you turn on echoing with **nocbreak()**. See Figure 9-6 and accompanying caution. After you turn echoing off, you can still echo characters with **addch()**.

EXAMPLE

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

NAME

cbreak() and **nocbreak()**

SYNOPSIS

```
#include < curses.h >
int cbreak()
int nocbreak()
```

NOTES

- **cbreak()** turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- **nocbreak()** returns to normal 'line at a time' processing. This is typically the initial setting.
- Neither returns any useful value.
- A **curses** program may not run properly if **cbreak()** is turned on and off within the same program or if the combination **nocbreak()** and **echo()** is used.
- See Figure 9-6 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under "curses Program Examples" in this chapter.

Controlling Output and Input

This page is intentionally left blank

Building Windows and Pads

An earlier section in this chapter, "More about **refresh()** and Windows" explained what windows and pads are and why you might want to use them. This section describes the **curses** routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch('c')** would become **waddch(mywin, 'c')** if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in "Getting Simple Output and Input."

- **waddch**(*win, ch*)
- **mvwaddch**(*win, y, x, ch*)
- **waddstr**(*win, str*)
- **mvwaddstr**(*win, y, x, str*)
- **wprintw**(*win, fmt [, arg...]*)
- **mvwprintw**(*win, y, x, fmt [, arg...]*)
- **wmove**(*win, y, x*)
- **wclear**(*win*) and **werase**(*win*)
- **wclrtoeol**(*win*) and **wclrtoeol**(*win*)
- **wrefresh**()

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the

Building Windows and Pads

addition of a *win* argument. Notice that the routines whose names begin with **mvw** take the *win* argument before the *y*, *x* coordinates, which is contrary to what the names imply. See **courses(3X)** for more information about these routines or the versions of the input routines **getch**, **getstr()**, and so on that you should use with windows.

All **w** routines can be used with pads except for **wrefresh()** and **wnoutrefresh()** (see below). In place of these two routines, you have to use **prefresh()** and **pnoutrefresh()** with pads.

The Routines **wnoutrefresh()** and **doupdate()**

If you recall from the earlier discussion about **refresh()**, we said that it sends the output from **stdscr** to the terminal screen. We also said that it was a macro that expands to **wrefresh(stdscr)** (see "What Every **courses** Program Needs" and "More about **refresh()** and Windows").

The **wrefresh()** routine is used to send the contents of a window (**stdscr** or one that you create) to a screen; it calls the routines **wnoutrefresh()** and **doupdate()**. Similarly, **prefresh()** sends the contents of a pad to a screen by calling **pnoutrefresh()** and **doupdate()**.

Using **wnoutrefresh()**—or **pnoutrefresh()** (this discussion will be limited to the former routine for simplicity)—and **doupdate()**, you can update terminal screens with more efficiency than using **wrefresh()** by itself. **wrefresh()** works by first calling **wnoutrefresh()**, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh()**, **wrefresh()** then calls **doupdate()**, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to a screen. However, by calling **wnoutrefresh()** for each window and then **doupdate()** only once, you can minimize the total number of characters transmitted and the

processor time used. The following sample program uses only one **doupdate()**:

```
#include <curses.h>

main()
{
    WINDOW *w1, *w2;

    initscr();
    w1 = newwin(2,6,0,3);
    w2 = newwin(1,4,5,4);
    waddstr(w1, "Bulls");
    wnoutrefresh(w1);
    waddstr(w2, "Eye");
    wnoutrefresh(w2);
    doupdate();
    endwin();
}
```

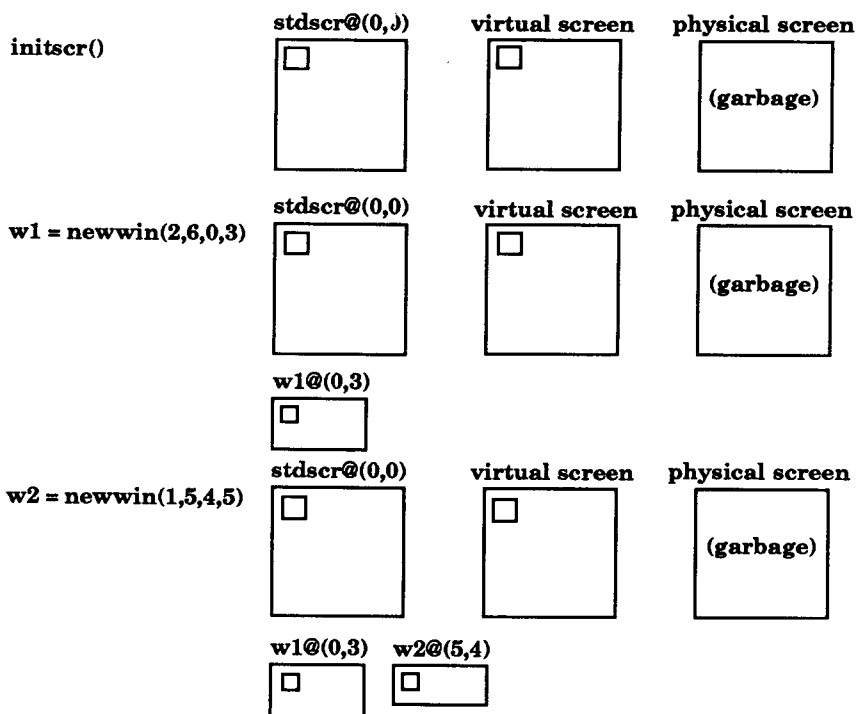
Notice from the sample that you declare a new window at the beginning of a **curses** program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named **w1** and **w2** with the routine **newwin()** according to certain specifications. **newwin()** is discussed in more detail below.

Figure 9-7 illustrates the effect of **wnoutrefresh()** and **doupdate()** on these two windows, the virtual screen, and the physical screen:

Building Windows and Pads



542

Figure 9-7: The Relationship Between a Window and a Terminal Screen
(Sheet 1 of 3)

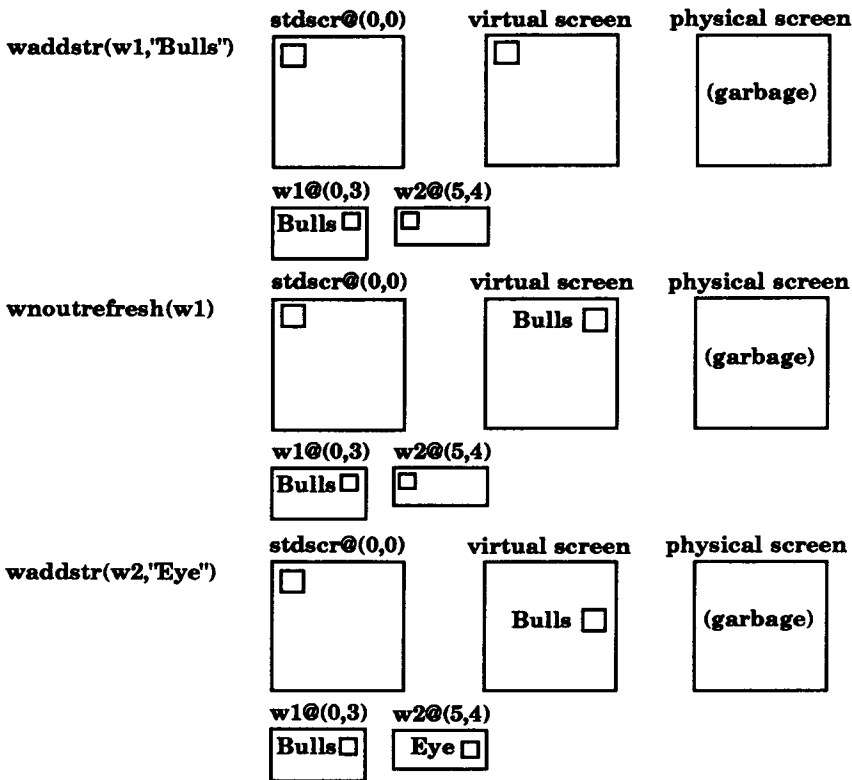
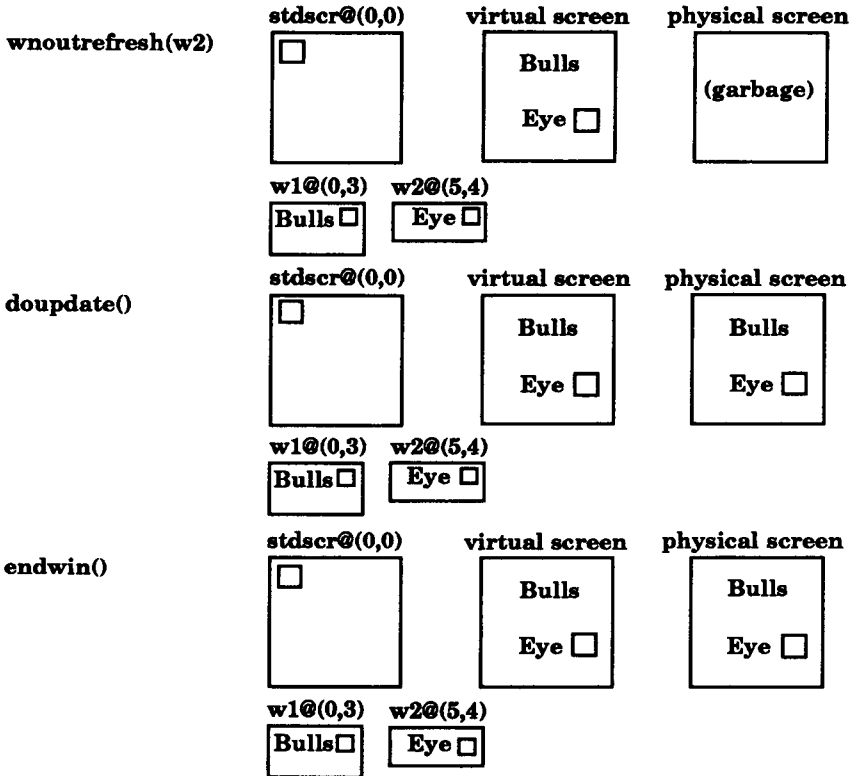


Figure 9-7: The Relationship Between a Window and a Terminal Screen (Sheet 2 of 3)

Building Windows and Pads



544

Figure 9-7: The Relationship Between a Window and a Terminal Screen (Sheet 3 of 3)

New Windows

Following are descriptions of the routines **newwin()** and **subwin()**, which you use to create new windows. For information about creating new pads with **newpad()** and **subpad()**, see the **curses(3X)** manual page.

NAME

newwin()

SYNOPSIS

```
#include <curses.h >
```

```
WINDOW * newwin(nlines, ncols, begin_y, begin_x)  
int nlines, ncols, begin_y, begin_x;
```

NOTES

- **newwin()** returns a pointer to a new window with a new data area.
- The variables **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

EXAMPLE

Recall the sample program using two windows; see Figure 9-7. Also see the **window** program under "curses Program Examples" in this chapter.

New Windows

NAME

subwin()

SYNOPSIS

#include < curses.h >

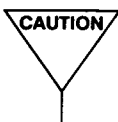
WINDOW * subwin(orig, nlines, ncols, begin_y, begin_x)

WINDOW * orig;

int nlines, ncols, begin_y, begin_x;

NOTES

- **subwin()** returns a new window that points to a section of another window, **orig**.
- **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.



Subwindows of subwindows do not work (as of the copyright date of this *System V Guide*).

EXAMPLE

```
#include <curses.h>

main()
{
    WINDOW *sub;

    initscr();
    box(stdscr, 'w', 'w'); /*See the curses(3X) manual */
                          /*page for box()          */
    mvwaddstr(stdscr, 7, 10, "----- this is 10,10");
    mvwaddch(stdscr, 8, 10, '|');
    mvwaddch(stdscr, 9, 10, 'v');
    sub = subwin(stdscr, 10, 20, 10, 10);
    box(sub, 's', 's');
    wnoutrefresh(stdscr);
    wrefresh(sub);
    endwin();
}
```

This program prints a border of **w**s around the **stdscr** (the sides of your terminal screen) and a border of **s**'s around the subwindow **sub** when it is run. For another example, see the **window** program under "**curses** Program Examples" in this chapter.

New Windows

This page is intentionally left blank

Using Advanced `curses` Features

Knowing how to use the basic `curses` routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The `curses` library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single `curses` program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the `curses(3X)` manual page before you try to use the advanced `curses` features.



The routines described under "Routines for Drawing Lines and Other Graphics" and "Routines for Using Soft Labels" are features that are new for UNIX System V Release 3.0. If a program uses any of these routines, it may not run on earlier releases of the UNIX system. You must use the Release 3.0 version of the `curses` library on UNIX System V Release 3.0 to work with these routines.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in `curses` programs. `curses` use the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in a `curses` program, you pass a set of variables whose names begin with `ACS_` to the `curses` routine `waddch()` or a related routine. For example, `ACS_ULCORNER` is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, `ACS_ULCORNER`'s value is the terminal's character for that glyph OR'd (|) with the bit-mask `A_ALTCHARSET`. If no line drawing character is available for that glyph, a standard ASCII character that approximates the glyph is

Using Advanced curses Features

stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a - (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_names and their defaults are listed on the **curses(3X)** manual page.

Part of an example program that uses line drawing characters follows. The example uses the **curses** routine **box()** to draw a box around a menu on a screen. **box()** uses the line drawing characters by default or when | (the pipe) and - are chosen. (See **curses(3X)**.) Up and down more indicators are drawn on the box border (using **ACS_UARROW** and **ACS_DARROW**) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);
...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
    waddch(menuwin, ACS_UARROW);
else
    addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
    waddch(menuwin, ACS_DARROW);
else
    waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET) )
    ACS_DARROW = 'V';
```

For more information, see **curses(3X)** in the *System V Reference Manual*.

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The **curses** library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a **curses** program to make use of them.

Let's briefly discuss most of the **curses** routines needed to use soft labels: **slk_clear**, **slk_restore**, **slk_init()**, **slk_set()**, **slk_refresh()** and **slk_noutrefresh()**.

When you use soft labels in a **curses** program, you have to call the routine **slk_int()** before **initscr()**. This sets an internal flag for **initscr()** to look at that says to use the soft labels. If **initscr()** discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels.

Using Advanced curses Features

The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The **curses** routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk_set()** takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk_noutrefresh()** is comparable to **wnoutrefresh()** in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh()** commonly follows, **slk_noutrefresh()** is the function that is most commonly used to output the labels.

Just as **wrefresh()** is equivalent to a **wnoutrefresh()** followed by a **doupdate()**, so too the function **slk_refresh()** is equivalent to a **slk_noutrefresh()** followed by a **doupdate()**.

To prevent the soft labels from getting in the way of a shell escape, **slk_clear()** may be called before doing the **endwin()**. This clears the soft labels off the screen and does a **doupdate()**. The function **slk_restore()** may be used to restore them to the screen. See the **curses(3X)** manual page for more information about the routines for using soft labels.

Working with More than One Terminal

A **curses** program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the **curses** library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking **\$TERM** in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

A **curses** program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

Using Advanced curses Features

References to terminals in a **curses** program have the type **SCREEN ***. A new terminal is initialized by calling **newterm**(*type*, *outfd*, *infd*). **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio**(3S) file pointer (**FILE ***) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr**(), which calls **newterm**(**getenv**("TERM"), **stdout**, **stdin**).

To change the current terminal, call **set_term**(*sp*) where *sp* is the screen reference to be made current. **set_term**() returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**(). Options such as **cbreak**() and **noecho**() must be set separately for each terminal. The functions **endwin**() and **refresh**() must be called separately for each terminal. Figure 9-8 shows a typical scenario to output a message to several terminals.

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 9-8: Sending a Message to Several Terminals

See the **two** program under "curses Program Examples" in this chapter for a more complete example.

Working with terminfo Routines

Some programs need to use lower level routines (i.e., primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines: the higher level **curses** routines make your program more portable to other UNIX systems and to a wider class of terminals.

555

NOTE

You are discouraged from using **terminfo** routines except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Figure 9-9.

```
#include <curses.h>
#include <term.h>
...
setupterm( (char*)0, 1, (int*)0 );
...
putp(clear_screen);
...
reset_shell_mode();
exit(0);
```

Figure 9-9: Typical Framework of a **terminfo** Program

The header files **<curses.h>** and **<term.h>** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm()** takes care of initialization. Passing this routine the values **(char *)0**, **1**, and **(int *)0** invokes reasonable defaults. If **setupterm()** can't figure out what kind of terminal you are on, it prints an error message and exits. **reset_shell_mode()** performs functions similar to **endwin()** and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm()**. It can be output using the **terminfo** routines **putp()** or **tputs()**, which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf(3S)**, because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch()** and **getch()** are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo(4)**; see **curses(3X)** for a list of all the **terminfo** routines.

Compiling and Running a terminfo Program

The general command line for compiling and the guidelines for running a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections "Compiling a **curses** Program" and "Running a **curses** Program" in this chapter for more information.

An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see "curses Program Examples") that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

Working with terminfo Routines

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;    /* Currently underlining */

main(argc, argv)
    int argc;
    char **argv;
{
    FILE *fd;
    int c, c2;
    int outch();

    if (argc > 2)
    {
        fprintf(stderr, "Usage: termhl [file]\n");
        exit(1);
    }

    if (argc == 2)
    {
        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
            perror(argv[1]);
            exit(2);
        }
    }
    else
    {
        fd = stdin;
    }
    setupterm((char*)0, 1, (int*)0);

    for (;;)
    {
        c = getc(fd);
        if (c == EOF)
```

(continued on next page)

```
break;
if (c == '\\')
{
    c2 = getc(fd);
    switch (c2)
    {
        case 'B':
            tputs(enter_bold_mode, 1, outch);
            continue;
        case 'U':
            tputs(enter_underline_mode, 1, outch);
            ulmode = 1;
            continue;
        case 'N':
            tputs(exit_attribute_mode, 1, outch);
            ulmode = 0;
            continue;
    }
    putch(c);
    putch(c2);
}
else
    putch(c);
}
fclose(fd);
fflush(stdout);
resetterm();
exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
int c;
{
    outch(c);
    if (ulmode && underline_char)
    {
        outch('\b');
        tputs(underline_char, 1, outch);
    }
}
```

(continued on next page)

Working with terminfo Routines

```
}  
  
/*  
 * Outchar is a function version of putchar that can be passed to  
 * tputs as a routine to call.  
 */  
outch(c)  
    int c;  
{  
    putchar(c);  
}
```

Let's discuss the use of the function `tputs(cap, affcnt, outch)` in this program to gain some insight into the **terminfo** routines. `tputs()` applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the **terminfo** database probably contain strings like `$ <20 >`, which means to pad for 20 milliseconds (see the following section "Specify Capabilities" in this chapter). `tputs` generates enough pad characters to delay for the appropriate time.

`tput()` has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention `affcnt` is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since `affcnt` is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, `affcnt` is always 1 and `outch` always calls `putchar`. For these programs, the routine `putp(cap)` is a convenient abbreviation. `termhl` could be simplified by using `putp()`.

Now to understand why you should use the **curses** level routines instead of **terminfo** level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low level details such as this are precisely why the **curses** level is recommended over the **terminfo** level. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** (see **curses(3X)**) could have been used instead of directly outputting

```
enter_bold_mode,  
enter_underline_mode,  
and  
exit_attribute_mode.
```

In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

This page is intentionally left blank

Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some UNIX system tools, like **vi(1)**, can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

The Virtual Terminal Interface (VTI) on the SUPERMAX has the effect that different terminals are able to use the same entry in the **terminfo** database. The entry **T3-24-C80** describes a terminal with 24 lines, 80 columns, cursor movement capabilities, insert and delete capabilities, and four attributes, (e.g. inverse video, underlining, blink and alternative intensity). This entry is the standard entry for terminals on a SUPERMAX computer. It will therefore rarely be necessary to build your own entry in the **terminfo** database.

For further information please refer to the *Virtual Terminal Interface Guide*.

Writing Terminal Descriptions

Descriptions of many popular terminals are already described in the **terminfo** database. However, it is possible that you'll want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.

Working with the terminfo Database

3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier.

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named "myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (|). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a **-w**) version of our fictitious terminal would be described as **myterm-w. term(5)** describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.
- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways – type:

```
stty2 line 0 - vtin - vtant - ;echo;cat - vu
```

Type in the keys you want to test;

for example, see what right arrow (→) transmits.

```
<CR>
```

```
<CTRL-D>
```

```
stty2 echo line 1 - vtin - vtant
```

Working with the terminfo Database

- The first line in this testing method sets up the terminal to carry out the tests. The **<CTRL-D>** helps return the terminal to its normal settings.
- See the **terminfo(4)** manual page. It lists all the capability names you have to use in a terminal description. The following section, "Specify Capabilities," gives details.

Specify Capabilities

Once you know the capabilities of your terminal, you have to describe them in your terminal description. You describe them with a string of comma-separated fields that contain the abbreviated **terminfo** name and, in some cases, the terminal's value for each capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as **bel = ^G**.

The list of capabilities may continue onto multiple lines as long as white space (that is, tabs and spaces) begins every line but the first of the description. Comments can be included in the description by putting a **#** at the beginning of the line.

The **terminfo(4)** manual page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled "Capname."

NOTE

For a **curses** program to run on any given terminal, its description in the **terminfo** database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

- # This shows a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as **cols#80**.
- = This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:
 - ^ This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as **^G**.
 - \E or \e These characters followed by another character show an escape instruction. An entry of **\EC** would transmit to the terminal as **ESCAPE - C**.
 - \n These characters provide a **<NL>** character sequence.
 - \l These characters provide a linefeed character sequence.
 - \r These characters provide a return character sequence.

Working with the terminfo Database

- `\t` These characters provide a tab character sequence.
- `\b` These characters provide a backspace character sequence.
- `\f` These characters provide a formfeed character sequence.
- `\s` These characters provide a space character sequence.
- `\nnn` This is a character whose three-digit octal is *nnn*, where *nnn* can be one to three digits.
- `$ < >` These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the "less than/greater than" symbols (`<` `>`). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as `$<20*>`. See the **terminfo(4)** manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (`.`) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

```
.bel = ^G,
```

With this background information about specifying capabilities, let's add the capability string to our description of `myterm`. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is **^G** (**bel**).
- An 80-column wide screen (**cols**).
- A 30-line long screen (**lines**).
- Use of xon/xoff protocol (**xon**).

By combining the name string (see the section "Name the Terminal") and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
      am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal `myterm` has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A **<CR>** is a CTRL-M (**cr**).
- A cursor up one line motion is a CTRL-K (**cuu1**).
- A cursor down one line motion is a CTRL-J (**cudl1**).

Working with the terminfo Database

- Moving the cursor to the left one space is a CTRL-H (**cub1**).
- Moving the cursor to the right one space is a CTRL-L (**cuf1**).
- Entering reverse video mode is an ESCAPE-D (**smso**).
- Exiting reverse video mode is an ESCAPE-Z (**rmso**).
- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).
- A terminal scrolls when receiving a <NL> at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
  am, bel=^G, cols#80, lines#30, xon,
  cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
  smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

570

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up arrow key generates an ESCAPE-[A (**kcuu1**).
- The down arrow key generates an ESCAPE-[B (**kcud1**).

- The right arrow key generates an ESCAPE - | C (**kcuf1**).
- The left arrow key generates an ESCAPE - | D (**kcub1**).
- The home key generates an ESCAPE - | H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
am, bel='G, cols#80, lines#30, xon,
cr='M, cuul='K, cudl='J, cubl='H, cuf1='L,
smso=ED, rmso=\EZ, el=\EK$<3>, ind=0
kbs='H, kcuul=\E[A, kcudl=\E[B, kcuf1=\E[C,
kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters — for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes (**sgr**) capabilities, are passed arguments in a **terminfo** program by the **tparm()** routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **printf(3S)** statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr**, takes nine arguments, one for each of the nine video attributes. See **terminfo(4)** for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE - [and followed with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18,

Working with the terminfo Database

from (0,0), the sequence 'ESCAPE - [6 ; 19 H' would be output.

Integer arguments are pushed onto the stack with a '%p' sequence followed by the argument number, such as '%p2' to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a '%d' sequence is used, exactly as in **printf**. Our terminal's **cup** sequence is built up as follows:

cup =	Meaning
\E[output ESCAPE - [
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
am, bel=^G, cols#80, lines#30, xon,
cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
kcubl=\E[D, khome=\E[H,
cup=\E[%i%p1%d;%p2%dH,
```

See **terminfo(4)** for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of **myterm** would be in a source file named **myterm.ti**. The compiled description of **myterm** would usually be placed in **/usr/lib/terminfo/m/myterm**, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm**, for example, to **/usr/lib/terminfo/f/fancy**. If the environment variable **\$TERMINFO** were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the **\$TERMINFO** directory. All programs using the entry would then look in the new directory for the description file if **\$TERMINFO** were set, before looking in the default **/usr/lib/terminfo**. The general format for the **tic** compiler is as follows:

```
tic [-v] [-c] file
```

The **-v** option causes the compiler to trace its actions and output information about its progress. The **-c** option causes a check for errors; it may be combined with the **-v** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **cat(1)** to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

```
tic -v myterm.ti <CR>
```

(The trace information appears as the compilation proceeds.)

Refer to the **tic(1M)** manual page in the *System V Reference Manual* for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable **\$TERMINFO** to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert line padding by commenting out **xon** in the description and then editing (using **vi(1)**) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput(1)** command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the **tput** command is as follows:

```
tput [-Ttype] capname
```

The type of terminal you are requesting information about is identified with the **-Ttype** option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **\$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

```
tput clear  
(The screen is cleared.)
```

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols  
(The number of columns used by the terminal appears here.)
```

The **tput(1)** manual page found in the *System V Reference Manual* contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp(1M)** command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420
```

compares the old and new 5420 entries.

To print out the **terminfo** source for the 5420, type

```
infocmp -I 5420
```

Converting a termcap Description to a terminfo Description



The **terminfo** database is designed to take the place of the **termcap** database. Because of the many programs and processes that have been written with and for the **termcap** database, it is not feasible to do a complete cutover at one time. Any conversion from **termcap** to **terminfo** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **captoinfo(1M)** command converts **termcap(4)** descriptions to **terminfo(4)** descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example,

```
captoinfo /etc/termcap
```

converts the file **/etc/termcap** to **terminfo** source, preserving comments and other extraneous information within the file. The command line

```
captoinfo
```

looks up the current terminal in the **termcap** database, as specified by the **\$TERM** and **\$TERMCAP** environment variables and converts it to **terminfo**.

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp -C** to get the **termcap** descriptions.

If you have been using cursor optimization programs with the **-ltermcap** or **-ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **-lcurses** option.

curses Program Examples

The following examples demonstrate uses of **curses** routines.

The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move()**, **mvaddstr()**, **flash()**, **wnoutrefresh()** and **clrtoeol()** routines. These routines are all discussed in this chapter under "Working with **curses** Routines."

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch()** routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch()**, **delch()**, **insertln()**, and **deleteln()** routines. These functions insert and delete a character or line. See **curses(3X)** for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

Examples

NOTE

Because not all terminals have arrow keys, your **curses** programs will work on more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, another important point is that the input command is terminated by CTRL-D, not the escape key. It is very tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **curses** programs, avoid the escape key.

```
579
/* editor: A screen-oriented editor. The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

#include <curses.h>
#define CTRL(c) ((c) & 037)
main(argc, argv)
int argc;
char **argv;
{
    extern void perror(), exit();
    int i, n, l;
    int c;
    int line = 0;
    FILE *fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s file\n", argv[0]);
        exit(1);
    }
    fd = fopen(argv[1], "r");
    if (fd == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    initscr();
    cbreak();
    nonl();
    noecho();
}
```

(continued on next page)

Examples

```

        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);
        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
            if (c == '\n')
                line++;
            if (line > LINES - 2)
                break;
            addch(c);
        }
        fclose(fd);
        move(0,0);
        refresh();
        edit();
        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l = 0; l < LINES - 1; l++)
        {
            n = len(l);
            for (i = 0; i < n; i++)
                putc(mvinch(l, i) & A_CHARTEXT, fd);
            putc('\n', fd);
        }
        fclose(fd);
        endwin();
        exit(0);
}
len(lineno)
int lineno;
{
    int linelen = COLS - 1;
    while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
        linelen--;
    return linelen + 1;
}
/* Global value of current cursor position */
int row, col;
edit()

```

(continued on next page)

```
{
    int c;
    for (;;)
    {
        move(row, col);
        refresh();
        c = getch();
        /* Editor commands */
        switch (c)
        {
            /* hjkl and arrow keys: move cursor
             * in direction indicated */
            case 'h':
            case KEY_LEFT:
                if (col > 0)
                    col--;
                else
                    flash();
                break;
            case 'j':
            case KEY_DOWN:
                if (row < LINES - 1)
                    row++;
                else
                    flash();
                break;
            case 'k':
            case KEY_UP:
                if (row > 0)
                    row--;
                else
                    flash();
                break;
            case 'l':
            case KEY_RIGHT:
                if (col < COLS - 1)
                    col++;
                else
                    flash();
                break;
        }
    }
}
```

(continued on next page)

Examples

```
/* i: enter input mode */
case KEY_IC:
case 'i':
    input();
    break;

/* x: delete current character */
case KEY_DC:
case 'x':
    delch();
    break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
    move(++row, col = 0);
    insertln();
    input();
    break;

/* d: delete current line */
case KEY_DL:
case 'd':
    deleteln();
    break;

/* ^L: redraw screen */
case KEY_CLEAR:
case CTRL('L'):
    wrefresh(curscr);
    break;

/* w: write and quit */
case 'w':
    return;

/* q: quit without writing */
case 'q':
    endwin();
    exit(2);

default:
    flash();
    break;
}
}
```

(continued on next page)

```
/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC
 */
input()
{
    int c;
    standout();
    mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;)
    {
        c = getch();
        if (c == CTRL('D') || c == KEY_EIC)
            break;
        insch(c);
        move(row, ++col);
        refresh();
    }
    move(LINES - 1, COLS - 20);
    clrtoeol();
    move(row, col);
    refresh();
}
```

Examples

This page is intentionally left blank

The highlight Program

This program illustrates a use of the routine `attrset()`. `highlight` reads a text file and uses embedded escape sequences to control attributes. `\U` turns on underlining, `\B` turns on bold, and `\N` restores the default output attributes.

Note the first call to `scrollok()`, a routine that we have not previously discussed (see `curses(3X)`). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, `scrollok()` automatically scrolls the terminal up a line and calls `refresh()`.

```
585
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
    FILE *fd;
    int c, c2;
    void exit(), perror();
    if (argc != 2)
    {
        fprintf(stderr, "Usage: highlight file\n");

        exit(1);
    }
    fd = fopen(argv[1], "r");
    if (fd == NULL)
    {
```

(continued on next page)

The highlight Program

```
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);
    nonl();
    while ((c = getc(fd)) != EOF)
    {
        if (c == '\\')
        {
            c2 = getc(fd);
            switch (c2)
            {
                case 'B':
                    attrset(A_BOLD);
                    continue;
                case 'U':
                    attrset(A_UNDERLINE);
                    continue;
                case 'N':
                    attrset(0);
                    continue;
            }
            addch(c);
            addch(c2);
        }
        else
            addch(c);
    }
    fclose(fd);
    refresh();
    endwin();
    exit(0);
}
```

The scatter Program

This program takes the first **LINES** - 1 lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```

/*
 *      The scatter program.
 */

#include <curses.h>
#include <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS 160
char s[MAXLINES][MAXCOLS];          /* Screen Array */
int T[MAXLINES][MAXCOLS];          /* Tag Array - Keeps track of
 * the number of characters
 * printed and their positions. */

main()
{
    register int row = 0,col = 0;
    register int c;
    int char_count = 0;
    time_t t;
    void exit(), srand();

    initscr();
    for(row = 0;row < MAXLINES;row++)
        for(col = 0;col < MAXCOLS;col++)
            s[row][col]=' ';

    col = row = 0;
    /* Read screen in */
    while ((c=getchar()) != EOF && row < LINES ) {

```

(continued on next page)

The scatter Program

```
        if(c != '\n')
        {
            /* Place char in screen array */
            s[row][col++] = c;
            if(c != ' ')
                char_count++;
        }
        else
        {
            col = 0;
            row++;
        }
    }

    time(&t); /* Seed the random number generator */
    srand((unsigned)t);

    while (char_count)
    {
        row = rand() % LINES;
        col = (rand() >> 2) % COLS;
        if (T[row][col] != 1 && s[row][col] != ' ')
        {
            move(row, col);
            addch(s[row][col]);
            T[row][col] = 1;
            char_count--;
            refresh();
        }
    }
    endwin();
    exit(0);
}
```

The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak()** so that you can depress the space bar without having to hit return; it calls **noecho()** to prevent the space from echoing on the screen. The **nonl()** routine is called to enable more cursor optimization. The **idlok()** routine is called to allow insert and delete line. (See **courses(3X)** for more information about these routines). Also notice that **clrtoeol()** and **clrtobot()** are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **courses()** program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }

    if ((fd=fopen(argv[1], "r")) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
}
```

(continued on next page)

The show Program

```
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for (line = 0; line < LINES; line++)
        {
            if (!fgets(linebuf, sizeof linebuf, fd))
            {
                clrtoeol();
                done();
            }
            move(line, 0);
            printw("%s", linebuf);
        }
        refresh();
        if (getch() == 'q')
            done();
    }

    void done()
    {
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
    }
}
```

The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "**sleep 100000**" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>
SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
    void done(), exit();
    unsigned sleep();
    char *getenv();
    int c;
    if (argc != 4)
    {
        fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
        exit(1);
    }
    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "wt");
    signal(SIGINT, done); /* die gracefully */
```

(continued on next page)

The two Program

```
me = newterm(getenv("TERM"), stdout, stdin); /* initialize my tty */
you = newterm(argv[2], fdyou, fdyou); /* Initialize the other terminal */
set_term(me); /* Set modes for my terminal */
noecho(); /* turn off tty echo */
cbreak(); /* enter cbreak mode */
nonl(); /* Allow linefeed */
nodelay(stdscr, TRUE); /* No hang on input */
set_term(you); /* Set modes for other terminal */
noecho();
cbreak();
nonl();
nodelay(stdscr, TRUE);

/* Dump first screen full on my terminal */
dump_page(me);

/* Dump second screen full on the other terminal */
dump_page(you);

for (;;) /* for each screen full */
{
    set_term(me);
    c = getch();
    if (c == 'q') /* wait for user to read it */
        done();
    if (c == ' ')
        dump_page(me);
    set_term(you);
    c = getch();
    if (c == 'q') /* wait for user to read it */
        done();
    if (c == ' ')
        dump_page(you);
    sleep(1);
}
}
dump_page(term)
SCREEN *term;
{
    int line;
    set_term(term);
    move(0, 0);
```

(continued on next page)


```
for (line = 0; line < LINES - 1; line++) {
    if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
        clrtoeb();
        done();
    }
    mvaddstr(line, 0, linebuf);
}
standout();
mvprintw(LINES - 1, 0, "--More--");
standend();
refresh();          /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES - 1, 0); /* to lower left corner */
    clrtoeol();        /* clear bottom line */
    refresh();         /* flush out everything */
    endwin();          /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES - 1, 0); /* to lower left corner */
    clrtoeol();        /* clear bottom line */
    refresh();         /* flush out everything */
    endwin();          /* curses cleanup */
    exit(0);
}
```

The two Program

This page is intentionally left blank

The window Program

This example program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh()** for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh()** on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin()** routine (which we have not discussed — see **courses(3X)**) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin()** for the new window to get it completely written out.

```
#include <curses.h>
WINDOW *cmdwin;
main()
{
    int i, c;
    char buf[120];
    void exit();

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0); /* top 3 lines */
    for (i = 0; i < LINES; i++)
        mvprintw(i, 0, "This is line %d of stdscr", i);

    for (;;)
    {
```

(continued on next page)

The window Program

```
refresh();
c = getch();
switch (c)

{

case 'c': /* Enter command from keyboard */
    werase(cmdwin);
    wprintw(cmdwin, "Enter command:");
    wmove(cmdwin, 2, 0);
    for (i = 0; i < COLS; i++)
        waddch(cmdwin, '-');
    wmove(cmdwin, 1, 0);
    touchwin(cmdwin);
    wrefresh(cmdwin);
    wgetstr(cmdwin, buf);
    touchwin(stdscr);

    /*
     * The command is now in buf.
     * It should be processed here.
     */

case 'q':
    endwin();
    exit(0);

}

}
```

Chapter 10: The Common Object File Format (COFF)

	Page
The Common Object File Format (COFF)	10 – 1
Definitions and Conventions	10 – 3
Sections	10 – 3
Physical and Virtual Addresses	10 – 4
Target Machine	10 – 4
File Header	10 – 5
Magic Numbers	10 – 5
Flags	10 – 6
File Header Declaration	10 – 7
Optional Header Information	10 – 7
Standard UNIX System a.out Header	10 – 8
Optional Header Declaration	10 – 9
Section Headers	10 – 10
Flags	10 – 11
Section Header Declaration	10 – 12
.bss Section Header	10 – 12
Sections	10 – 13
Relocation Information	10 – 13
Relocation Entry Declaration	10 – 15
Line Numbers	10 – 16
Line Number Declaration	10 – 17
Symbol Table	10 – 18
Special Symbols	10 – 20
Inner Blocks	10 – 21

Table of Contents

	Page
Symbols and Functions.....	10 - 23
Symbol Table Entries.....	10 - 24
Symbol Names.....	10 - 24
Storage Classes	10 - 26
Storage Classes for Special Symbols	10 - 28
Symbol Value Field.....	10 - 29
Section Number Field	10 - 30
Section Numbers and Storage Classes	10 - 31
Type Entry	10 - 32
Type Entries and Storage Classes	10 - 35
Structure for Symbol Table Entries	10 - 37
Auxiliary Table Entries.....	10 - 38
Auxiliary Entry Declaration	10 - 44
String Table	10 - 47
Access Routines	10 - 48

The Common Object File Format (COFF)

The COFF format described in this chapter covers MOTOROLA 680X0. COFF for MIPS is an extension to the format shown.

This section describes the Common Object File Format (COFF) used on SUPERMAX computers with the UNIX operating system. COFF is the format of the output file produced by the assembler, **as**, and the link editor, **ld**.

Some key features of COFF are

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains

- a file header
- optional header information
- a table of section headers
- data corresponding to the section headers

The Common Object File Format (COFF)

- relocation information
- line numbers
- a symbol table
- a string table

Figure 10-1 shows the overall structure.

FILE HEADER
Optional Information
Section 1 Header
...
Section <i>n</i> Header
Raw Data for Section 1
...
Raw Data for Section <i>n</i>
Relocation Info for Sect. 1
...
Relocation Info for Sect. <i>n</i>
Line Numbers for Sect. 1
...
Line Numbers for Sect. <i>n</i>
SYMBOL TABLE
STRING TABLE

Figure 10-1: Object File Format

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the `-s` option of the `ld` command, or if the line number information, symbol table, and string table are removed by the `strip` command. The line number information does not appear unless the program is compiled with the `-g` option of the `cc` command. Also, if there are no

unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the UNIX operating system loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

NOTE

It is a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

The Common Object File Format (COFF)

Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; but in all versions of COFF on UNIX systems, the physical address is equivalent to the virtual address.

Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular cpu. The term target machine refers to the cpu on which the object file is destined to run.

File Header

The file header contains the 20 bytes of information shown in Figure 10-2. The last 2 bytes are flags that are used by `ld` and object file utilities.

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number
2-3	unsigned short	f_nscns	Number of sections
4-7	long int	f_timdat	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see Figure 10-3)

Figure 10-2: File Header Contents

Magic Numbers

The magic number specifies the target machine on which the object file is executable.

The Common Object File Format (COFF)

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file `filehdr.h`, and are shown in Figure 10-3.

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file.
F_EXEC	00002	File is executable (i.e., no unresolved external references).
F_LNNO	00004	Line numbers stripped from the file.
F_LSYMS	00010	Local symbols stripped from the file.
F_MINMAL	00020	Not used by the UNIX system.
F_UPDATE	00040	Not used by the UNIX system.
F_SWABD	00100	Not used by the UNIX system.
F_AR16WR	00200	File has the byte ordering used by the PDP*-11/70 processor.
F_AR32WR	00400	File has the byte ordering used by the VAX-11/780 (i.e., 32 bits per word, least significant byte first).
F_AR32W	01000	File has the byte ordering used by the SUPERMAX computers (i.e., 32 bits per word, most significant byte first).
F_PATCH	02000	Not used by the UNIX system.
F_BM32ID	0160000	WE 32000 processor ID field.

Figure 10-3: File Header Flags

The Common Object File Format (COFF)

File Header Declaration

The C structure declaration for the file header is given in Figure 10-4. This declaration may be found in the header file **filehdr.h**.

```
struct filehdr
{
    unsigned short  f_magic; /* magic number */
    unsigned short  f_nscns; /* number of section */

    long           f_timdat; /* time and date stamp */

    long           f_symptr; /* file ptr to symbol table */

    long           f_nsyms; /* number entries in the symbol table */

    unsigned short  f_opthdr; /* size of optional header */

    unsigned short  f_flags; /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

Figure 10-4: File Header Declaration

Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information.

General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field **f_opthdr**.

The Common Object File Format (COFF)

Standard UNIX System a.out Header

By default, files produced by the link editor for a UNIX system always have a standard UNIX system **a.out** header in the optional header field.

Bytes	Declaration	Name	Description
0 - 1	short	magic	Magic number
2 - 3	short	vstamp	Version stamp
4 - 7	long int	tsize	Size of text in bytes
8 - 11	long int	dsize	Size of initialized data in bytes
12 - 15	long int	bsize	Size of uninitialized data in bytes
16 - 19	long int	entry	Entry point
20 - 23	long int	text_start	Base address of text
24 - 27	long int	data_start	Base address of data

Figure 10-5: Optional Header Contents

Whereas, the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the UNIX operating system are given in Figure 10-6.

Value	Meaning
0407	The text segment is not write-protected or sharable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write protected.
0413	Text and data segments are aligned within a.out so it can be directly paged.

Figure 10-6: UNIX System Magic Numbers

Optional Header Declaration

The C language structure declaration currently used for the UNIX system **a.out** file header is given in Figure 10-7. This declaration may be found in the header file **aouthdr.h**.

```
typedef struct aouthdr
{
    short    magic;           /* magic number */
    short    vstamp;         /* version stamp */
    long     tsize;          /* text size in bytes, padded */
                                /* to full word boundary */
    long     dsize;          /* initialized data size */
    long     bsize;          /* uninitialized data size */
    long     entry;          /* entry point */
    long     text_start;     /* base of text for this file */
    long     data_start      /* base of data for this file */
} AOUTHDR;
```

Figure 10-7: **aouthdr** Declaration

The Common Object File Format (COFF)

Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 10-8.

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Figure 10-9)

Figure 10-8: Section Header Contents

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the UNIX system function **fseek(3S)**.

The Common Object File Format (COFF)

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 10-9.

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For <code>.lib</code> section (treated like STYP_INFO)

Figure 10-9: Section Header Flags

Section Header Declaration

The C structure declaration for the section headers is described in Figure 10-10. This declaration may be found in the header file `scnhdr.h`.

```
struct scnhdr
{
    char    s_name[8];        /* section name */
    long    s_paddr;         /* physical address */
    long    s_vaddr;         /* virtual address */
    long    s_size;          /* section size */
    long    s_scnptr;        /* file ptr to section raw data */

    long    s_relptr;        /* file ptr to relocation */

    long    s_lnnoptr;       /* file ptr to line number */

    unsigned short s_nreloc; /* number of relocation entries */

    unsigned short s_nlnno; /* number of line number entries */

    long    s_flags;         /* flags */

};

#define SCNHDR  struct scnhdr
#define SCNHSZ  sizeof(SCNHDR)
```

Figure 10-10: Section Header Declaration

.bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a `.bss` section. A `.bss` section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a `.bss` section has no relocation entries, no line number entries, and no data. Therefore, a `.bss` section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as

well as all file pointers in a **.bss** section header, are 0. The same is true of the **STYP_NOLOAD** and **STYP_DSECT** sections.

Sections

Figure 10-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor **SECTIONS** directives (see Chapter 11) allow users to, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections

If no **SECTIONS** directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a **.text** section, are linked together the output object file contains a single **.text** section made up of the combined input **.text** sections.

Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 10-11.

The Common Object File Format (COFF)

Bytes	Declaration	Name	Description
0-3	long int	r_vaddr	(Virtual) address of reference
4-7	long int	r_symndx	Symbol table index
8-9	unsigned short	r_type	Relocation type

Figure 10-11: Relocation Section Contents

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Figure 10-12.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address.
R_DIR32S	012	Direct 32-bit reference to the symbol's virtual address, with the 32-bit value stored in the reverse order in the object file.

Figure 10-12: Relocation Types

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 10-13. This declaration may be found in the header file **reloc.h**.

```
struct reloc
{
    long          r_vaddr; /* virtual address of reference */
    long          r_symndx; /* index into symbol table */
    unsigned short r_type; /* relocation type */
};

#define RELOC    struct reloc
#define RELSZ    10
```

Figure 10-13: Relocation Entry Declaration

The Common Object File Format (COFF)

Line Numbers

When invoked with the `-g` option, the `cc` command cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like `sdb`. All line numbers in a section are grouped by function as shown in Figure 10-14.

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

Figure 10-14: Line Number Grouping

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function, and appear in increasing order of address.

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 10-15.

```
struct lineno
{
    union
    {
        long    l_symndx; /* symtbl index of func name */
        long    l_paddr; /* paddr of line number */
    } l_addr;
    unsigned short l_lnno; /* line number */
};

#define LINENO    struct lineno
#define LINESZ    6
```

Figure 10-15: Line Number Entry Declaration

The Common Object File Format (COFF)

Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 10-16.

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
...
statics
...
filename 2
function 1
local symbols for function 1
...
statics
...
defined global symbols
undefined global symbols

Figure 10-16: COFF Symbol Table

The word `static` in Figure 10-16 means symbols defined with the C language storage class `static` outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

The Common Object File Format (COFF)

Special Symbols

The symbol table contains some special symbols that are generated by **as**, and other tools. These symbols are given in Figure 10-17.

Symbol	Meaning
.file	filename
.text	address of .text section
.data	address of .data section
.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section .text
edata	next available address after the end of the output section .data
end	next available address after the end of the output section .bss

Figure 10-17: Special Symbols in the Symbol Table

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.xfake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is **.xfake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces, {, and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol, **.bb**, is put in the symbol table immediately before the first local symbol of that block. Also a special symbol, **.eb**, is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 10-18.

```

.bb
-----
local symbols
for that block
-----
.eb

```

Figure 10-18: Special Symbols (**.bb** and **.eb**)

Because inner blocks can be nested by several levels, the **.bb**–**.eb** pairs and associated symbols may also be nested. See Figure 10-19.

The Common Object File Format (COFF)

```
{                                     /* block 1 */
  int i;
  char c;
  ...
  {                                     /* block 2 */
    long a;
    ...
    {                                     /* block 3 */
      int x;
      ....
    }                                     /* block 3 */
  }                                     /* block 2 */
  {                                     /* block 4 */
    long i;
    ...
  }                                     /* block 4 */
}                                     /* block 1 */
```

Figure 10-19: Nested blocks

The symbol table would look like Figure 10-20 on the following page.

The Common Object File Format (COFF)

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Figure 10-20: Example of the Symbol Table

Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 10-21.

function name
.bf
local symbol
.ef

Figure 10-21: Symbols for Functions

The Common Object File Format (COFF)

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 10-22. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Declaration	Name	Description
0-7	(see text below)	_n	These 8 bytes contain either a symbol name or an index to a symbol
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_scnm	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	Storage class of symbol
17	char	n_numaux	Number of auxiliary entries

Figure 10-22: Symbol Table Entry Format

Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4

The Common Object File Format (COFF)

bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 10-23.

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	Zero in this field indicates the name is in the string table
4-7	long	n_offset	Offset of the name in the string table

Figure 10-23: Name Field

Special symbols generated by the C Compilation System are discussed above in "Special Symbols."

The Common Object File Format (COFF)

Storage Classes

The storage class field has one of the values described in Figure 10-24. These `#define`'s may be found in the header file `storclass.h`.

Mnemonic	Value	Storage Class
<code>C_EFCN</code>	-1	physical end of a function
<code>C_NULL</code>	0	-
<code>C_AUTO</code>	1	automatic variable
<code>C_EXT</code>	2	external symbol
<code>C_STAT</code>	3	static
<code>C_REG</code>	4	register variable
<code>C_EXTDEF</code>	5	external definition
<code>C_LABEL</code>	6	label
<code>C_ULABEL</code>	7	undefined label
<code>C_MOS</code>	8	member of structure
<code>C_ARG</code>	9	function argument
<code>C_STRTAG</code>	10	structure tag
<code>C_MOU</code>	11	member of union
<code>C_UNTAG</code>	12	union tag
<code>C_TPDEF</code>	13	type definition
<code>C_USTATIC</code>	14	uninitialized static
<code>C_ENTAG</code>	15	enumeration tag
<code>C_MOE</code>	16	member of enumeration
<code>C_REGPARM</code>	17	register parameter

Mnemonic	Value	Storage Class
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	filename
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

Figure 10-24: Storage Classes

All of these storage classes except for C_ALIAS and C_HIDDEN are generated by the **cc** or **as** commands. The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility (described in the *System V Reference Manual*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C_HIDDEN is not used by any UNIX system tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

The Common Object File Format (COFF)

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Figure 10-25.

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Figure 10-25: Storage Class by Special Symbols

Also some storage classes are used only for certain special symbols.

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Figure 10-26: Restricted Storage Classes

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Figure 10-27.

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARAM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size

The Common Object File Format (COFF)

Storage Class	Meaning of Value
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

Figure 10-27: Storage Class and Value

If a symbol has storage class `C_FILE`, the value of that symbol equals the symbol table entry index of the next `.file` symbol. That is, the `.file` entries form a one-way linked list in the symbol table. If there are no more `.file` entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Figure 10-28.

Mnemonic	Section Number	Meaning
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

Figure 10-28: Section Number

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued

symbols include automatic and register variables, function arguments, and `.eos` symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (i.e., FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0 and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the `.bss` section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

629

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 10-29.

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG

The Common Object File Format (COFF)

Storage Class	Section Number
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Figure 10-29: Section Number and Storage Class

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is

The Common Object File Format (COFF)

d6	d5	d4	d3	d2	d1	typ
-----------	-----------	-----------	-----------	-----------	-----------	------------

Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Figure 10-30.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_VOID	1	void
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Figure 10-30: Fundamental Types

The Common Object File Format (COFF)

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Figure 10-31.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Figure 10-31: Derived Types

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Figure 10-32 shows the type entries that are legal for each storage class.

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_BLOCK	no	no	no	T_NULL

The Common Object File Format (COFF)

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_FCN	no	no	no	T_NULL
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Figure 10-32: Type Entries by Storage Class

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 10-33. This declaration may be found in the header file `syms.h`.

```

struct syment
{
    union
    {
        char        _n_name[SYMNMLEN];    /* symbol name*/
        struct
        {
            long     _n_zeroes;           /* symbol name */

            long     _n_offset;           /* location in string table */
        } _n_n;
        char        *_n_nptr[2];         /* allows overlaying */
    } _n;
    unsigned long   n_value;             /* value of symbol */

    short           n_scnm;              /* section number */

    unsigned short  n_type;              /* type and derived */

    char            n_sclass;            /* storage class */

    char            n_numaux;            /* number of aux entries */
};

#define n_name      _n._n_name
#define n_zeroes   _n._n_n._n_zeroes
#define n_offset   _n._n_n._n_offset
#define n_nptr     _n._n_nptr[1]

#define SYMNMLEN  8
#define SYMESZ    18    /* size of a symbol table entry */

```

Figure 10-33: Symbol Table Entry Declaration

The Common Object File Format (COFF)

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 10-34.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	filename
.text, .data, .bss	C_STAT	DT_NON	T_NULL	section
<i>tagname</i>	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
<i>fname</i>	C_EXT C_STAT	DT_FCN	(Note 1)	function
<i>arrname</i>	(Note 2)	DT_ARY	(Note 1)	array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Figure 10-34: Auxiliary Symbol Table Entries

Notes to Figure 10-34:

1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In Figure 10-34, *tagname* means any symbol name including the special symbol *xfake*, and *fname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Figure 10-34 should have a union format in its auxiliary entry.

NOTE It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available, and should be obtained from the **n_numaux** field in the symbol table.

Filenames

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Figure 10-35.

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-17	—	—	unused (filled with zeroes)

Figure 10-35: Format for Auxiliary Table Entries for Sections

The Common Object File Format (COFF)

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 10-36.

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, and enumeration
8-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	-	-	unused (filled with zeroes)

Figure 10-36: Tag Names Table Entries

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 10-37:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Figure 10-37: Table Entries for End of Structures

Functions

The auxiliary table entries for functions have the format shown in Figure 10-38:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_lnnoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of the function's address in the transfer vector table (not used in UNIX system)

Figure 10-38: Table Entries for Functions

The Common Object File Format (COFF)

Arrays

The auxiliary table entries for arrays have the format shown in Figure 10-39. Defining arrays having more than four dimensions produces a warning message.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_inno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	-	-	unused (filled with zeroes)

Figure 10-39: Table Entries for Arrays

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 10-40:

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_inno	C-source line number
6-17	-	-	unused (filled with zeroes)

Figure 10-40: End of Block and Function Entries

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 10-41:

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_lno	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	-	-	unused (filled with zeroes)

Figure 10-41: Format for Beginning of Block and Function

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Figure 10-42:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Figure 10-42: Entries for Structures, Unions, and Enumerations

The Common Object File Format (COFF)

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example:

```
typedef struct people STUDENT;
struct people
{
    char name[20];
    long id;
};
typedef struct people EMPLOYEE;
```

The symbol **EMPLOYEE** has an auxiliary table entry in the symbol table but symbol **STUDENT** will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 10-43 as follows. This declaration may be found in the header file **syms.h**.

The Common Object File Format (COFF)

```

union auxent
{
    struct
    {
        long    x_tagndx;
        union
        {
            struct
            {
                unsigned short  x_lnno;
                unsigned short  x_size;
            } x_lnsz;
            long    x_fsize;
        } x_misc;
        union
        {
            struct
            {
                long    x_lnnoptr;
                long    x_endndx;
            } x_fcn;

            struct
            {
                unsigned short  x_dimen[DIMNUM];
            } x_ary;
        } x_fcary;
        unsigned short  x_tvndx;
    } x_sym;

    struct
    {
        char    x_fname[FILNMLEN];
    } x_file;

    struct
    {
        long    x_scrlen;
        unsigned short  x_nreloc;
        unsigned short  x_nlinno;
    }

```

(continued on next page)

The Common Object File Format (COFF)

```

    } x_scn;

    struct
    {
        long    x_tvfill;
        unsigned short  x_tvlen;
        unsigned short  x_tvran[2];
    } x_tv;
}
#define FILNMLEN 14
#define DIMNUM 4
#define AUXENT union auxent
#define AUXESZ 18

```

Figure 10-43: Auxiliary Symbol Table Entry

String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer than eight characters, **long_name_1** and **another_one**) the string table has the format as shown in Figure 10-44:

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

Figure 10-44: String Table

The index of **long_name_1** in the string table is 4 and the index of **another_one** is 16.

The Common Object File Format (COFF)

Access Routines

UNIX system releases contain a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of the *System V Reference Manual*. A summary of what is available can be found in the *System V Reference Manual* under **ldfcn(4)**.

Chapter 11: The Link Editor

	Page
The Link Editor.....	11- 1
Memory Configuration.....	11- 2
Sections	11- 2
Addresses	11- 3
Binding.....	11- 3
Object File.....	11- 3
Link Editor Command Language.....	11- 5
Expressions	11- 5
Assignment Statements.....	11- 6
Specifying a Memory Configuration.....	11- 8
Section Definition Directives	11-10
File Specifications	11-11
Load a Section at a Specified Address	11-13
Aligning an Output Section	11-14
Grouping Sections Together	11-14
Creating Holes Within Output Sections.....	11-17
Creating and Defining Symbols at Link-Edit Time.....	11-19
Allocating a Section Into Named Memory	11-21
Initialized Section Holes or .bss Sections.....	11-21
Notes and Special Considerations.....	11-25
Changing the Entry Point.....	11-25
Use of Archive Libraries.....	11-26
Dealing With Holes in Physical Memory.....	11-28
Allocation Algorithm.....	11-29
Incremental Link Editing.....	11-30

Table of Contents

	Page
DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections .	11 - 32
Output File Blocking	11 - 34
Nonrelocatable Input Files.....	11 - 34
Syntax Diagram for Input Directives	11 - 37

The Link Editor

The link editor described in this chapter covers MOTOROLA 680X0.

The link editor for MIPS R3000 differs a bit.

In Chapter 2 there was a discussion of link editor command line options (some of which may also be provided on the `cc(1)` command line). This chapter contains information on the Link Editor Command Language.

The command language enables you to

- specify the memory configuration of the target machine
- combine the sections of an object file in arrangements other than the default
- bind sections to specific addresses or within specific portions of memory
- define or redefine global symbols

Under most normal circumstances there is no compelling need to have such tight control over object files and where they are located in memory. When you do need to be very precise in controlling the link editor output, you do it by means of the command language.

Link editor command language directives are passed in a file named on the `ld(1)` command line. Any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives. The following paragraphs define terms and describe conditions with which you need to be familiar before you begin to use the command language.

Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K-1 are then not configured. Unconfigured memory is treated as reserved or unusable by `ld(1)`. Nothing can ever be linked into unconfigured memory. Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) illegal or nonexistent with respect to the linking process. Memory configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc. are with respect to the configured sections of the address space.

Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address.

While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the **ld(1)** command language.

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by **ld(1)**. **ld(1)** accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to **ld(1)** can also be absolute files.

Files produced from the compilation system may contain, among others, sections called **.text** and **.data**. The **.text** section contains the instruction text (executable instructions), **.data** contains initialized data variables.

The Link Editor

For example, if a C program contained the global (i.e., not inside a function) declaration

```
int i = 100;
```

and the assignment

```
i = 0;
```

then compiled code from the C assignment is stored in **.text**, and the variable **i** is located in **.data**.

Link Editor Command Language

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 11-2, "Syntax Diagram for Input Directives.") Constants are as in C with a number recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers's. Symbol names may contain uppercase or lowercase letters, digits, and the underscore, `_`. Symbols within an expression have the value of the address of the symbol only. `ld(1)` does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

`ld(1)` uses a `lex`-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names reserved and unavailable as symbol names or section names:

ADDR	COMMON	LENGTH	OVERLAY	SIZEOF
ALIGN	COPY	MEMORY	PHY	SPARE
ASSIGN	DSECT	NEXT	RANGE	TV
BIND	GROUP	NOLOAD	REGIONS	
BLOCK	INFO	ORIGIN	SECTIONS	
addr	block	length	origin	sizeof
align	group	next	phy	spare
assign	l	o	range	
bind	len	org	s	

The operators that are supported, in order of precedence from high to low, are shown in Figure 11-1:

symbol
! ~ - (UNARY Minus)
* / %
+ - (BINARY Minus)
>> <<
== != > < <= >=
&
&&
= += -= *= /=

Figure 11-1: Operator Symbols

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

654

Assignment Statements

External symbols may be defined and assigned addresses via the assignment statement. The syntax of the assignment statement is

```
symbol = expression;
```

or

```
symbol op= expression;
```

where *op* is one of the operators +, -, *, or /. Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the actual relocation of the text

and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to **ld(1)**.

Assignment statements are normally placed outside the scope of section-definition directives (see "Section Definition Directives" under "Link Editor Command Language"). However, there exists a special symbol, called **dot**, **.**, that can occur only within a section-definition directive. This symbol refers to the current address of **ld(1)**'s location counter. Thus, assignment expressions involving **.** are evaluated during the allocation phase of **ld(1)**. Assigning a value to the **.** symbol within a section-definition directive can increment (but not decrement) **ld(1)**'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of the **.** symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an n - byte boundary within an output section, where n is a power of 2. For example, the expression

align(n)

is equivalent to

$(. + n - 1) \& \sim (n - 1)$

SIZEOF and **ADDR** are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside of section directives.

Link editor expressions may have either an absolute or a relocatable value. When **ld(1)** creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

655

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.
- The difference of two relocatable symbols from the same section is absolute.
- All other expressions are combinations of the above.

Specifying a Memory Configuration

MEMORY directives are used to specify

1. The total size of the virtual space of the target machine.
2. The configured and unconfigured areas of the virtual space.

If no directives are supplied, **ld(1)** assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters \$, ., or _. Names of memory ranges are used by **ld(1)** only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld(1)**'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory.

As an option on the MEMORY directive, attributes may be associated with a named memory area. In future releases this may be used to provide error checking. Currently, error checking of this type is not implemented.

The attributes currently accepted are

1. R : readable memory
2. W : writable memory
3. X : executable, i.e., instructions may reside in this memory
4. I : initializable, i.e., stack areas are typically not initialized

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

The syntax of the MEMORY directive is

```
MEMORY
{
    name1 (attr) :      origin = n1, length = n2
    name2 (attr) :      origin = n3, length = n4
    etc.
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype.

The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, **ld(1)** can be told that memory is configured in some manner other than the default.

Link Editor Command Language

For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this.

```
MEMORY
{
    valid : org = 0x10000, len = 0xFE0000
}
```

Section Definition Directives

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections s1 and s2 and the other containing sections s3 and s4, the output object file contains the four sections s1, s2, s3, and s4.

The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is

```

SECTIONS
{
    secname1 :
    {
        file_specifications,
        assignment_statements
    }
    secname2 :
    {
        file_specifications,
        assignment_statements
    }
    etc.
}

```

The various types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

```

filename ( secname )
or
filename ( secnam1 secnam2 . . . )

```

Sections of an input file are separated either by white space or commas as are the file specifications themselves.

```

filename [COMMON]

```

may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

Link Editor Command Language

If a file name appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section. For example,

```
SECTIONS
{
    outsec1:
    {
        file1.o (sec1)
        file2.o
        file3.o (sec1, sec2)
    }
}
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be

1. section **sec1** from file **file1.o**
2. all sections from **file2.o**, in the order they appear in the file
3. section **sec1** from file **file3.o**, and then section **sec2** from file **file3.o**

If there are any additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in **file1.o** or **file3.o**, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The code

```
*(secname)
```

may be used to indicate all previously unallocated input sections of the given name, regardless of what input file they are contained in.

Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an **ld(1)** option as shown in the following SECTIONS directive example:

```
SECTIONS
{
    outsec addr:
    {
        . . .
    }
    etc.
}
```

The *addr* is the bonding address expressed as a C constant. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld(1)** issues an appropriate error message. *addr* may also be the word **BIND**, followed by a parenthesized expression. The expression may use the pseudo-functions **SIZEOF**, **ADDR** or **NEXT**. **NEXT** accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; **SIZEOF** and **ADDR** accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The **SECTIONS** directives defining output sections need not be given to **ld(1)** in any particular order, unless **SIZEOF** or **ADDR** is used.

ld(1) does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. **ld(1)** directives can be used to force a section to start on an odd byte boundary although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld(1)** issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an n -byte boundary, where n is a power of 2. The `ALIGN` option of the `SECTIONS` directive performs this function, so that the option

```
ALIGN(n)
```

is equivalent to specifying a bonding address of

```
( . + n - 1 ) & ~ ( n - 1 )
```

For example

```
SECTIONS
{
    outsec ALIGN(0x20000) :
    {
        . . .
    }
    etc.
}
```

The output section `outsec` is not bound to any given address but is placed at some virtual address that is a multiple of `0x20000` (e.g., at address `0x0`, `0x20000`, `0x40000`, `0x60000`, etc.).

Grouping Sections Together

The default allocation algorithm for `ld(1)`

1. Links all input `.init` sections together, followed by `.text` sections, into one output section. This output section is called `.text` and is bound to an address of `0x0` plus the size of all headers in the output file.

2. Links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine dependent constant plus a number dependent on the size of headers and text.
3. Links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated so as to immediately follow the output section **.data**. Note that the output section **.bss** is not given any particular address alignment.

Specifying any SECTIONS directives results in this default allocation not being performed. Rather than relying on the **ld(1)** default algorithm, if you are manipulating COFF files, the one certain way of determining address and order information is to take it from the file and section headers. The default allocation of **ld(1)** is equivalent to supplying the following directive:

```

SECTIONS
{
    .text sizeof_headers : { *(.init) *(.text) }
    GROUP BIND( NEXT(align_value) +
                ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
    {
        .data      : { }
        .bss       : { }
    }
}

```

where *align_value* is a machine dependent constant. The GROUP command ensures that the two output sections, **.data** and **.bss**, are allocated (e.g., grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

Link Editor Command Language

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following **SECTIONS** directive is used:

```

SECTIONS
{
    GROUP
    {
        .text      : { }
        .data      : { }
        .bss       : { }
    }
}
    
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the **GROUP** directive. To bind to **0xC0000**, use

```
GROUP 0xC0000 : {
```

To align to **0x10000**, use

```
GROUP ALIGN(0x10000) : {
```

With this addition, first the output section **.text** is bound at **0xC0000** (or is aligned to **0x10000**); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the **GROUP** directive is not used, each output section is treated as an independent entity:


```
SECTIONS
{
    .text : { }
    .data ALIGN(0x20000) : { }
    .bss  : { }
}
```

The **.text** section starts at virtual address 0x0 (if it is in configured memory) and the **.data** section at a virtual address aligned to 0x20000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section. The order in which output sections are defined to **ld(1)** cannot be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections

The special symbol dot, **.**, appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, **.** causes **ld(1)**'s location counter to be incremented or reset and a hole left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the **-f** option in "Using the Link Editor" and the discussion of filling holes in "Initialized Section Holes" or **.bss** Sections." in this chapter.

Consider the following section definition:

Link Editor Command Language

```

outsec:
{
    . += 0x1000;
    f1.o (.text)
    . += 0x100;
    f2.o (.text)
    . = align (4);
    f3.o (.text)
}

```

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section **f1.o (.text)** is linked after this hole.
2. The **.text** section of input file **f2.o** begins at 0x100 bytes following the end of **f1.o (.text)**.
3. The **.text** section of **f3.o** is linked to start at the next full word boundary following the **.text** section of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld(1)** treats the output section as if it began at address zero. As a result, if, in the above example, **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address—even though **f3.o (.text)** is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

It should be noted that the assembler, **as**, always pads the sections it generates to a full word length making explicit alignment specifications unnecessary. This also holds true for the compiler.

Expressions that decrement `.` are illegal. For example, subtracting a value from the location counter is not allowed since overwrites are not allowed. The most common operators in expressions that assign a value to `.` are `+=` and **align**.

Creating and Defining Symbols at Link-Edit Time

The assignment instruction of `ld(1)` can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. Use of `.` to adjust `ld(1)`'s location counter during allocation.
2. Use of `.` to assign an allocation-dependent value to a symbol.
3. Assigning an allocation-independent value to a symbol.

Case 1) has already been discussed in the previous section.

Case 2) provides a means to assign addresses (known only after allocation) to symbols. For example,

```

SECTIONS
{
    outsc1: {...}
    outsc2:
    {
        file1.o (s1)
        s2_start = . ;
        file2.o (s2)
        s2_end = . - 1;
    }
}
    
```

The symbol **s2_start** is defined to be the address of **file2.o(s2)**, and **s2_end** is the address of the last byte of **file2.o(s2)**.

Link Editor Command Language

Consider the following example:

```
SECTIONS
{
    outsc1:
    {
        file1.o (.data)
        mark = .;
        . += 4;
        file2.o (.data)
    }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file1.o**'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving **.** must appear within **SECTIONS** definitions since they are evaluated during allocation. Assignment instructions that do not involve **.** can appear within **SECTIONS** definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. The **ld(1)** issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a MEMORY directive). (The > notation is borrowed from the UNIX system concept of redirected output.) For example,

```

MEMORY
{
    mem1:          o=0x000000    l=0x10000
    mem2 (RW):     o=0x020000    l=0x40000
    mem3 (RW):     o=0x070000    l=0x40000
    mem1:          o=0x120000    l=0x04000
}

SECTIONS
{
    outsec1: { f1.o(.data) } > mem1
    outsec2: { f2.o(.data) } > mem3
}
    
```

This directs **ld(1)** to place **outsec1** anywhere within the memory area named **mem1** (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Creating Holes within Output Sections"), **ld(1)** normally puts out bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the **.o** file or a hole

Link Editor Command Language

in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized **.bss** section, if part of such a section is initialized, then the entire section is initialized. In other words, if a **.bss** section is to be combined with a **.text** or **.data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will hold:

- a. Explicit initialization options must be used to initialize all **.bss** sections in the output section.
- b. **ld(1)** will use the default fill value to initialize all **.bss** sections in the output section.

Consider the following **ld(1)** ifile:

```
SECTIONS
{
    sec1:
    {
        f1.o
        . += 0x200;
        f2.o (.text)
    } = 0xDFFF
    sec2:
    {
        f1.o (.bss)
        f2.o (.bss) = 0x1234
    }
    sec3:
    {
        f3.o (.bss)
        . . .
    } = 0xFFFF
    sec4: { f4.o (.bss) }
}
```

In the example above, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2**, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All **.bss** sections within **sec3** as well as all holes are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object file for this section.

This page is intentionally left blank

Notes and Special Considerations

Changing the Entry Point

The UNIX system **a.out** optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

- a. The value of the symbol specified with the **-e** option, if present, is used.
- b. The value of the symbol **_start**, if present, is used.
- c. The value of the symbol **main**, if present, is used.
- d. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the **-e** option or by using an assignment instruction in an ifile of the form

```
    _start = expression;
```

If **ld(1)** is called through **cc(1)**, a startup routine is automatically linked in. Then, when the program is executed, the routine **exit(1)** is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld(1)** directly or when changing the entry point. The user must supply the startup routine or make sure that the program always calls **exit** rather than falling through the end. Otherwise, the program will dump core.

Use of Archive Libraries

Each member of an archive library (e.g., **libc.a**) is a complete object file. Archive libraries are created with the **ar(1)** command from object files generated by **cc** or **as**. An archive library is always processed using selective inclusion: only those members that resolve existing undefined-symbol references are taken from the library for link editing. Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

- a. There exists a reference to a symbol defined in that member.
- b. The reference is found by **ld(1)** prior to the actual scanning of the library.

When a library member is included by searching the library inside a **SECTIONS** directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library outside of a **SECTIONS** directive, all input sections from the library member are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

1. Specific members of a library cannot be referenced explicitly in an ifile.
2. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The **-I** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the **-I** option by simply giving the (full or relative) UNIX system file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld(1)** will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

1. The input files **file1.o** and **file2.o** each contain a reference to the external function FCN.
2. Input **file1.o** contains a reference to symbol ABC.
3. Input **file2.o** contains a reference to symbol XYZ.
4. Library **liba.a**, member 0, contains a definition of XYZ.
5. Library **libc.a**, member 0, contains a definition of ABC.
6. Both libraries have a member 1 that defines FCN.

If the **ld(1)** command were entered as

```
ld file1.o -la file2.o -lc
```

then the FCN references are satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ remains undefined (because the library **liba.a** is searched before **file2.o** is specified). If the **ld(1)** command were entered as

```
ld file1.o file2.o -la -lc
```

then the FCN references is satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0. If the **ld(1)** command were entered as

```
ld file1.o file2.o -lc -la
```

then the FCN references is satisfied by **libc.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0.

Notes and Special Considerations

The `-u` option is used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

```
ld -u rout1 -la
```

creates an undefined symbol called `rout1` in `ld(1)`'s global symbol table. If any member of library `liba.a` defines this symbol, it (and perhaps other members as well) is extracted. Without the `-u` option, there would have been no unresolved references or undefined symbols to cause `ld(1)` to search the archive library.

Dealing With Holes in Physical Memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
    mem1:    o = 0x00000    l = 0x02000
    mem2:    o = 0x40000    l = 0x05000
    mem3:    o = 0x20000    l = 0x10000
}
```

Let the files `f1.o`, `f2.o`, . . . `fn.o` each contain three sections `.text`, `.data`, and `.bss`, and suppose the combined `.text` section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the `.text` output section so `ld(1)` may do allocation. For example,

```
SECTIONS
{
    txt1:
    {
        f1.o (.text)
        f2.o (.text)
        f3.o (.text)
    }
    txt2:
    {
        f4.o (.text)
        f5.o (.text)
        f6.o (.text)
    }
    etc.
}
```

Allocation Algorithm

An output section is formed either as a result of a `SECTIONS` directive, by combining input sections of the same name, or by combining `.text` and `.init` into `.text`. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. `ld(1)` uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit bonding addresses were specified are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any

Notes and Special Considerations

alignment taken into consideration.

3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no **SECTIONS** directives are given, then output sections are allocated in the order they appear to **ld(1)**. Otherwise, output sections are allocated in the order they were defined or made known to **ld(1)** into the first available space they fit.

Incremental Link Editing

As previously mentioned, the output of **ld(1)** can be used as an input file to subsequent **ld(1)** runs providing that the relocation information is retained (**-r** option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application. For example,

Step 1:

ld -r -o outfile1 ifile1 infile1.o

```
/* ifile1 */
SECTIONS
{
    ss1:
    {
        f1.o
        f2.o
        . . .
        fn.o
    }
}
```

Step 2:

ld -r -o outfile2 ifile2 infile2.o

```
/* ifile2 */
SECTIONS
{
    ss2:
    {
        g1.o
        g2.o
        . . .
        gn.o
    }
}
```

Step 3:

ld -a -o final.out outfile1 outfile2

By judiciously forming subsystems, applications may achieve a form of

Notes and Special Considerations

incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules

1. Intermediate link edits should contain only **SECTIONS** declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.
2. All allocation and memory directives, as well as any assignment statements, are included only in the final **ld(1)** call.

DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections

Sections may be given a type in a section definition as shown in the following example:

```

SECTIONS
{
    name1 0x200000 (DSECT)      : { file1.o }
    name2 0x400000 (COPY)      : { file2.o }
    name3 0x600000 (NOLOAD)    : { file3.o }
    name4          (INFO)      : { file4.o }
    name5 0x900000 (OVERLAY)   : { file5.o }
}
    
```

The **DSECT** option creates what is called a dummy section. A dummy section has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld(1)**.

2. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not as a DSECT).
4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from **file1.o** are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the COPY option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An INFO section is the same as a COPY section but its purpose is to carry information about the object file whereas the COPY section may contain valid text and data. INFO sections are usually used to contain file version identification information.

A section with the type of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, etc.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

Output File Blocking

The **BLOCK** option (applied to any output section or **GROUP** directive) is used to direct **ld(1)** to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
    .text BLOCK(0x200) : { }
    .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this **SECTIONS** directive, **ld(1)** assures that each section, **.text** and **.data**, is physically written at a file offset, which is a multiple of **0x200** (e.g., at an offset of 0, **0x200**, **0x400**, and so forth, in the file).

Nonrelocatable Input Files

If a file produced by **ld(1)** is intended to be used in a subsequent **ld(1)** run, the first **ld(1)** run should have the **-r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld(1)** does not have relocation or symbol table information (perhaps from the action of a **strip(1)** command, or from being link edited without a **-r** option or with a **-s** option), the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met.

Notes and Special Considerations

1. Each input file must have no unresolved external references.
2. Each input file must be bound to the exact same virtual address as it was bound to in the **ld(1)** run that created it.

NOTE

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld(1)**.

Syntax Diagram for Input Directives

NOTE

Two punctuation symbols, square brackets and curly braces, do double duty in the following diagram.

Where the actual symbols, [] and { } are used, they are part of the syntax and must be present when the directive is specified.

Where you see the symbols [and] (larger and in bold), it means the material enclosed is optional.

Where you see the symbols { and } (larger and in bold), it means multiple occurrences of the material enclosed are permitted.

Directives	Expanded Directives
< ifile >	{ < cmd > }
< cmd >	< memory > < sections > < assignment > < filename > < flags >
< memory >	MEMORY { < memory_spec > { [,] < memory_spec > } }
< memory_spec >	< name > [< attributes >] : < origin_spec > [,] < length_spec >
< attributes >	({ R W X I })
< origin_spec >	< origin > = < long >
< lenth_spec >	< length > = < long >
< origin >	ORIGIN o org origin
< length >	LENGTH l len length

Syntax Diagram for Input Directives

Directives	Expanded Directives
<sections>	SECTIONS { { <sec_or_group> } }
<sec_or_group>	<section> <group> <library>
<group>	GROUP <group_options> : {
<section_list>	<section_list> } [<mem_spec>]
<section>	<section> { [,] <section> }
	<name> <sec_options> :
	{ <statement> }
	[<fill>] [<mem_spec>]
<group_options>	[<addr>] [<align_option>] [<block_option>]
<sec_options>	[<addr>] [<align_option>]
	[<block_option>] [<type_option>]
<addr>	<long> <bind>(<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN align
<block_option>	<block> (<long>)
<block>	BLOCK block
<type_option>	(DSECT) (NOLOAD) (COPY) (INFO) (OVERLAY)
<fill>	= <long>
<mem_spec>	> <name>
	> <attributes>
<statement>	<filename>
	<filename> (<name_list>) [COMMON]
	* (<name_list>) [COMMON
	<assignment>
	<library>
	<i>null</i>

Syntax Diagram for Input Directives

Directives	Expanded Directives
<name_list>	<section_name> [,] { <section_name> }
<library>	-l<name>
<bind>	BIND bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> .
<assign_op>	= += -= *= /= =
<end>	; ,
<expr>	<expr> <binary_op> <expr>
<binary_op>	<term> * / % + - >> << == != > < <= > = & &&
<term>	<long> <name> <align> (<term>) (<expr>) <unary_op> <term> <phy> (<lside>) <sizeof>(<sectionname>) <next>(<long>) <addr>(<sectionname>)
<unary_op>	! -
<phy>	PHY phy
<sizeof>	SIZEOF sizeof

Syntax Diagram for Input Directives

Directives	Expanded Directives
< next >	NEXT next
< addr >	ADDR addr
< flags >	- e < wht_space > < name > - f < wht_space > < long > - h < wht_space > < long > - l < name > - m - o < wht_space > < filename > - r - s - t - u < wht_space > < name > - z - H - L < path_name > - M - N - S - V - VS < wht_space > < long > - a - x

Syntax Diagram for Input Directives

Directives	Expanded Directives
< name >	Any valid symbol name
< long >	Any valid long integer constant
< wht_space >	Blanks, tabs, and newlines
< filename >	Any valid UNIX operating system filename. This may include a full or partial path name.
< sectionname >	Any valid section name, up to 8 characters.
< path_name >	Any valid UNIX operating system path name (full or partial).

Figure 11-2: Syntax Diagram For Input Directives

Syntax Diagram for Input Directives

This page is intentionally left blank