IRIS-4D Programmer's Guide

Volume II

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0601-010

IRIS-4D Programmer's Guide

Volume II

Version 1.0

Document Number 007-0601-010

Technical Publications:

Marcia Allen
Kathleen Chaix

Special Thanks to the Technical Marketing Group

IRIS-4D Programmer's Guide, Volume II
Version 1.0
Document Number 007-0601-010

Silicon Graphics, Inc.
Mountain View, California

Contents

Introduction

Purpose xiii
- Audience and Prerequisite Knowledge xiii
- Organization xiii
- The C Connection xiv
- Hardware/Software Dependencies xiv
- Notation Conventions xiv
- Command References xv
- Information in the Examples xvi

1. Programming in a UNIX System Environment

The UNIX System Environment 1-1
- The UNIX Idea 1-1
- The UNIX Philosophy 1-3

UNIX System Tools 1-4
- Tools Covered and Not Covered in This Guide 1-4
- The Shell as a Prototyping Tool 1-5

Three Programming Environments 1-6
- Single-User Programming 1-6
- Application Programming 1-7
- Systems Programming 1-7

Summary 1-8

2. Programming Basics

Programming in a UNIX Environment 2-1

Choosing a Programming Language 2-2
- Supported Languages in the UNIX Environment 2-2
  - C Language 2-3
  - FORTRAN 2-4
  - Assembly Language 2-4
- Special-Purpose Languages 2-4
  - awk 2-4
  - lex 2-5
  - yacc 2-5
  - M4 2-6
  - bc and dc 2-6

curses 2-6
Compiling and Link Editing 2-7
Compiling C Programs 2-7
Compiling FORTAN Programs 2-7
Compiler Diagnostic Messages 2-7
Link Editing 2-8
The UNIX System/Language Interface 2-9
Using C to Illustrate the Interface 2-9
Passing Arguments to a Program 2-9
System Calls and Subroutines 2-13
Categories of System Calls and Subroutines 2-13
Where the Manual Pages Can Be Found 2-20
Using System Calls and Subroutines in C Programs 2-20
Header Files and Libraries 2-26
Object File Libraries 2-27
Input and Output 2-28
Three Files to Handle Standard I/O Streams 2-28
Named Files 2-29
The UNIX System and Low-level I/O 2-30
System Calls for Environment or Status Information 2-30
Processes 2-32
system(3S) 2-33
exec(2) 2-33
fork(2) 2-34
Pipes 2-36
Error Handling 2-37
Signals and Interrupts 2-38
Analysis and Debugging 2-40
Sample Program 2-40
cflow 2-45
ctrace 2-48
cxref 2-52
lint 2-58
prof 2-59
size 2-61
strip 2-61
Program-Organizing Utilities 2-62
The make Command 2-62
The Archive File 2-63

Use of SCCS by Single-User Programmers 2-69

3. Application Programming

Application Programming Objectives 3-1
Application Environment Characteristics 3-2
Numbers 3-2
Portability 3-2
Documentation 3-2
Language Selection 3-4
Influences 3-4
Special-Purpose Languages 3-5
The **awk** Utility 3-5
Using **awk** 3-6
The **lex** and **yacc** Utilities 3-6
Using **lex** 3-7
Using **yacc** 3-8
Advanced Programming Tools 3-10
Memory Management 3-10
File and Record Locking 3-11
How File and Record Locking Works 3-12
lockf 3-14
Interprocess Communications 3-14
IPC **get** Calls 3-15
IPC **ctl** Calls 3-15
IPC **op** Calls 3-16
Programming Terminal Screens 3-16
curses 3-17
Programming Support Tools 3-18
Link Edit Command Language 3-18
Common Object File Format 3-18
Libraries 3-19
The Object File Library 3-19
Common Object File Interface Macros (**ldfcn.h**) 3-22
The Math Library 3-23
A Basic Lesson on Debugging 3-26
How Does **edge** Work? 3-26
About the edge Environment 3-27
lint as a Portability Tool 3-27
Project Control Tools 3-29
make 3-29

SCCS 3-30
liber, A Library System 3-32

4. awk
An Overview of **awk** Programming 4-1
Basic **awk** 4-2
Program Structure 4-2
Using **awk** 4-3
Fields 4-3
Printing 4-4
Formatted Printing 4-5
Simple Patterns 4-6
Simple Actions 4-7
Built-in Variables 4-7
User-defined Variables 4-8
Functions 4-8
Useful One-Line **awk** Programs 4-8
Error Messages 4-10
Patterns 4-11
BEGIN and **END** 4-11
Relational Expressions 4-12
Regular Expressions 4-13
Combinations of Patterns 4-16
Pattern Ranges 4-17
Actions 4-18
Built-in Variables 4-18
Arithmetic 4-18
Strings and String Functions 4-21
Field Variables 4-24
Number or String? 4-25
Control Flow Statements 4-26
Arrays 4-28
User-Defined Functions 4-31
Comments 4-31
Output 4-32
The **print** Statement 4-32
Output Separators 4-32
The **printf** Statement 4-33
Output into Files 4-34
Output into Pipes 4-35

Input 4-37
Files and Pipes 4-37
Input Separators 4-37
Multi-Line Records 4-38
The **getline** Function 4-38
Command-line Arguments 4-40
Using awk with other Commands 4-42
The **system** Function 4-42
Cooperation with the Shell 4-42
Example Applications 4-45
Generating Reports 4-45
Additional Examples 4-47
awk Summary 4-50
Command Line 4-50
Patterns 4-50
Control Flow Statements 4-50
Input-Output 4-51
String Functions 4-51
Arithmetic Functions 4-52
Operators (increasing precedence) 4-52
Regular expressions (increasing precedence) 4-53
Built-in Variables 4-53
Limits 4-54
Initialization, Comparison, and Type Coercion 4-54

5. lex
An Overview of **lex** Programming 5-1
Writing **lex** Programs 5-3
The Fundamental **lex** Rules 5-3
Specifications 5-3
Actions 5-5
Advanced **lex** Usage 5-6
Some Special Features 5-7
Definitions 5-11
Subroutines 5-12
Using **lex** with **yacc** 5-13
Running **lex** Under the UNIX System 5-16

6. yacc
An Overview of **yacc** Programming 6-1

Basic Specifications 6-4
Actions 6-6
Lexical Analysis 6-9
Parser Operation 6-12
Ambiguity and Conflicts 6-17
Precedence 6-22
Error Handling 6-26
The **yacc** Environment 6-30
Hints for Preparing Specifications 6-32
Input Style 6-32
Left Recursion 6-32
Lexical Tie-Ins 6-33
Reserved Words 6-35
Advanced **yacc** Features 6-36
Simulating **error** and **accept** in Actions 6-36
Accessing Values in Enclosing Rules 6-36
Support for Arbitrary Value Types 6-38
yacc Input Syntax 6-39
Examples 6-43
1. A Simple Example 6-43
2. An Advanced Example 6-47

7. File and Record Locking
An Overview of File and Record Locking 7-1
Terminology 7-2
File Protection 7-4
Opening a File for Record Locking 7-4
Setting a File Lock 7-5
Setting and Removing Record Locks 7-8
Getting Lock Information 7-13
Deadlock Handling 7-16
Selecting Advisory or Mandatory Locking 7-17
Mandatory Locking 7-18
Record Locking and Future UNIX Releases 7-18

8. Interprocess Communication
An Overview of Inter-Process Communication 8-1
Messages 8-2
Getting Message Queues 8-6
Using **msgget** 8-7

Example Program 8-12
Controlling Message Queues 8-16
Using **msgctl** 8-16
Example Program 8-18
Operations for Messages 8-24
Using **msgop** 8-24
Example Program 8-26
Semaphores 8-37
Using Semaphores 8-39
Getting Semaphores 8-42
Using **semget** 8-42
Example Program 8-46
Controlling Semaphores 8-50
Using **semctl** 8-51
Example Program 8-52
Operations on Semaphores 8-64
Using **semop** 8-64
Example Program 8-66
Shared Memory 8-72
Using Shared Memory 8-73
Getting Shared Memory Segments 8-77
Using **shmget** 8-77
Example Program 8-81
Controlling Shared Memory 8-85
Using **shmctl** 8-85
Example Program 8-87
Operations for Shared Memory 8-96
Using **shmop** 8-96
Example Program 8-97

9. curses/terminfo
The Terminal Information Utilities Package 9-1
What is **curses**? 9-2
What is **terminfo**? 9-3
How **curses** and **terminfo** Work Together 9-5
Other Components of the Terminal Information Utilities 9-5
Working with **curses** Routines 9-7
What Every **curses** Program Needs 9-7
The Header File **<curses.h>** 9-7
The Routines **initscr()**, **refresh()**, **endwin()** 9-8

Compiling a **curses** Program 9-10
Running a **curses** Program 9-10
More about **initscr()** and Lines and Columns 9-11
More about **refresh()** and Windows 9-11
Getting Simple Output and Input 9-13
Output 9-13
Input 9-26
Controlling Output and Input 9-34
Output Attributes 9-34
Bells, Whistles, and Flashing Lights 9-38
Input Options 9-39
Building Windows and Pads 9-44
Output and Input 9-44
The Routines **wnoutrefresh()** and **doupdate()** 9-45
New Windows 9-46
Using Advanced **curses** Features 9-50
Routines for Drawing Lines and Other Graphics 9-50
Routines for Using Soft Labels 9-52
Working with More than One Terminal 9-53
Working with **terminfo** Routines 9-55
What Every **terminfo** Program Needs 9-55
Compiling and Running a **terminfo** Program 9-57
An Example **terminfo** Program 9-57
Working with the **terminfo** Database 9-62
Writing Terminal Descriptions 9-62
Name the Terminal 9-62
Learn About the Capabilities 9-63
Specify Capabilities 9-64
Compile the Description 9-69
Test the Description 9-70
Comparing or Printing **terminfo** Descriptions 9-71
Converting a **termcap** Description to
a **terminfo** Description 9-72
curses Program Examples 9-73
The **editor** Program 9-73
The **highlight** Program 9-81
The **scatter** Program 9-83
The **show** Program 9-86
The **two** Program 9-88
The **window** Program 9-91

10. make

An Overview of the **make** Utility 10-1
Basic Features 10-2
Description Files and Substitutions 10-7
  Comments 10-7
  Continuation Lines 10-7
  Macro Definitions 10-7
  General Form 10-7
  Dependency Information 10-8
  Executable Commands 10-8
  Extensions of $*, $@, and $< 10-9
  Output Translations 10-9
The Recursive **Makefile** 10-11
  Suffixes and Transformation Rules 10-11
  Implicit Rules 10-11
  Archive Libraries 10-13
SCCS Filenames: the Tilde 10-16
  The Null Suffix 10-17
  include Files 10-18
  SCCS Makefiles 10-18
  Dynamic Dependency Parameters 10-18
Command Usage 10-20
  The **make** Command 10-20
  Environment Variables 10-21
Suggestions and Warnings 10-23
Internal Rules 10-24

11. Source Code Control System (SCCS)

The Source Code Control System 11-1
SCCS for Beginners 11-2
  Terminology 11-2
  Creating an SCCS File via **admin** 11-2
  Retrieving a File via **get** 11-3
  Recording Changes via **delta** 11-4
  Additional Information about **get** 11-4
  The **help** Command 11-5
Delta Numbering 11-7
SCCS Command Conventions 11-10
  x.files and z.files 11-10
  Error Messages 11-11

SCCS Commands 11-12
The **get** Command 11-12
ID Keywords 11-13
Retrieval of Different Versions 11-14
Retrieval With Intent to Make a Delta 11-16
The **unget** Command 11-17
Additional **get** Options 11-17
Concurrent Edits of Different SID 11-18
Concurrent Edits of Same SID 11-21
Keyletters That Affect Output 11-21
The **delta** Command 11-23
The **admin** Command 11-25
Creation of SCCS Files 11-26
Inserting Commentary for the Initial Delta 11-26
Initialization and Modification of SCCS File Parameters 11-27
The **prs** Command 11-28
The **sact** Command 11-30
The **help** Command 11-30
The **rmdel** Command 11-31
The **cdc** Command 11-31
The **what** Command 11-32
The **sccsdiff** Command 11-33
The **comb** Command 11-33
The **val** Command 11-34
The **vc** Command 11-35
SCCS Files 11-36
Protection 11-36
Formatting 11-37
Auditing 11-38

12. lint
The **lint** Program 12-1
Using **lint** 12-2
lint Message Types 12-4
Unused Variables and Functions 12-4
Set/Used Information 12-5
Flow of Control 12-5
Function Values 12-6
Type Checking 12-7

Type Casts 12-8
Nonportable Character Use 12-8
Assignments of **longs** to **ints** 12-9
Strange Constructions 12-9
Old Syntax 12-10
Pointer Alignment 12-11
Multiple Uses and Side Effects 12-11

A. Index to Utilities A-1

Glossary G-1

An Overview of File and Record Locking

Mandatory and advisory file and record locking both are available on current releases of the UNIX system. The intent of this capability to is 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.

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.

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 database. This can be done easily by setting the set-group-ID bit (see **chmod**(1)) of the database 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:

```
#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 database 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:

```
#include <fcntl.h>
#define MAX_TRY10
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_TRY10
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 {
          .
          ./* data portion of 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;
           * 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".
             */
            (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.

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_IEXEC>>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
```

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 UNIX Releases

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.

An Overview of Inter-Process Communication

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.

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 "non-blocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- 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 msqid_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.

Once a uniquely identified message queue and data structure are created, message operations [**msgop()**] 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 *IRIS-4D Programmer's 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.

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
| ORed 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 *IRIS-4D Programmer's 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 **msqid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msqid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **msqid** is returned if the system call is successful.

Refer to the **msgget**(2) page in the *IRIS-4D Programmer's 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 *IRIS-4D Programmer's 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 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**. When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the message get, msgget(),
**system call capabilities.*/

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>
#include    <errno.h>

/*Start of main C language program*/
main()
{
    key_t key;            /*declare as long integer*/
    int opperm, flags;
    int msqid, opperm_flags;
    /*Enter the desired key*/
    printf("Enter the desired key in hex = ");
    scanf("%x", &key);


    /*Enter the desired octal operation
      permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);
```

Figure 8-4: **msgget()** System Call Example (Sheet 1 of 3)

```
/*Set the desired flags.*/
printf("\nEnter corresponding number to\n");
printf("set the desired flags:\n");
printf("No flags                    = 0\n");
printf("IPC_CREAT                   = 1\n");
printf("IPC_EXCL                    = 2\n");
printf("IPC_CREAT and IPC_EXCL      = 3\n");
printf("            Flags           = ");

/*Get the flag(s) to be set.*/
scanf("%d", &flags);

/*Check the values.*/
printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
    key, opperm, flags);

/*Incorporate the control fields (flags) with
  the operation permissions*/
switch (flags)
{
case 0:     /*No flags are to be set.*/
    opperm_flags = (opperm | 0);
    break;
case 1:     /*Set the IPC_CREAT flag.*/
    opperm_flags = (opperm | IPC_CREAT);
    break;
case 2:     /*Set the IPC_EXCL flag.*/
    opperm_flags = (opperm | IPC_EXCL);
    break;
case 3:     /*Set the IPC_CREAT and IPC_EXCL flags.*/
    opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
}
```

Figure 8-4: **msgget()** System Call Example (Sheet 2 of 3)

```
        /*Call the msgget system call.*/
        msqid = msgget (key, opperm_flags);

        /*Perform the following if the call is unsuccessful.*/
        if(msqid == -1)
        {
            printf ("\nThe msgget system call failed!\n");
            printf ("The error number = %d\n", errno);
        }

        /*Return the msqid upon successful completion.*/
        else
            printf ("\nThe msqid = %d\n", msqid);
        exit(0);
    }
```

Figure 8-4: **msgget**() System Call Example (Sheet 3 of 3)

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 *IRIS-4D Programmer's 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.
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 *IRIS-4D Programmer's 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 indentifier'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.

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**. When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the message control, msgctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include     <stdio.h>
#include     <sys/types.h>
#include     <sys/ipc.h>
#include     <sys/msg.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    int uid, gid, mode, bytes;
    int rtrn, msqid, command, choice;
    struct msqid_ds msqid_ds, *buf;
    buf = &msqid_ds;

    /*Get the msqid, and command.*/
    printf("Enter the msqid = ");
    scanf("%d", &msqid);
    printf("\nEnter the number for\n");
    printf("the desired command:\n");
    printf("IPC_STAT     =  1\n");
    printf("IPC_SET      =  2\n");
    printf("IPC_RMID     =  3\n");
    printf("Entry        =  ");
    scanf("%d", &command);
```

Figure 8-5: **msgctl()** System Call Example (Sheet 1 of 4)

```
/*Check the values.*/
printf ("\nmsqid =%d, command = %d\n",
    msqid, command);

switch (command)
{
case 1:    /*Use msgctl() to duplicate
      the data structure for
              msqid in the msqid_ds area pointed
              to by buf and then print it out.*/
    rtrn = msgctl(msqid, IPC_STAT,
        buf);
    printf ("\nThe USER ID = %d\n",
        buf->msg_perm.uid);
    printf ("The GROUP ID = %d\n",
        buf->msg_perm.gid);
    printf ("The operation permissions = 0%o\n",
        buf->msg_perm.mode);
    printf ("The msg_qbytes = %d\n",
        buf->msg_qbytes);
    break;
case 2:    /*Select and change the desired
              member(s) of the data structure.*/
    /*Get the original data for this msqid
          data structure first.*/
    rtrn = msgctl(msqid, IPC_STAT, buf);
    printf("\nEnter the number for the\n");
    printf("member to be changed:\n");
    printf("msg_perm.uid    = 1\n");
    printf("msg_perm.gid    = 2\n");
    printf("msg_perm.mode   = 3\n");
    printf("msg_qbytes      = 4\n");
    printf("Entry           = ");
```

Figure 8-5: **msgctl()** System Call Example (Sheet 2 of 4)

```
        scanf("%d", &choice);
        /*Only one choice is allowed per
          pass as an illegal entry will
              cause repetitive failures until
          msqid_ds is updated with
              IPC_STAT.*/

        switch(choice){
        case 1:
            printf("\nEnter USER ID = ");
            scanf ("%d", &uid);
            buf->msg_perm.uid = uid;
            printf("\nUSER ID = %d\n",
                buf->msg_perm.uid);
            break;
        case 2:
            printf("\nEnter GROUP ID = ");
            scanf("%d", &gid);
            buf->msg_perm.gid = gid;
            printf("\nGROUP ID = %d\n",
                buf->msg_perm.gid);
            break;
        case 3:
            printf("\nEnter MODE = ");
            scanf("%o", &mode);
            buf->msg_perm.mode = mode;
            printf("\nMODE = 0%o\n",
                buf->msg_perm.mode);
            break;
```

Figure 8-5: **msgctl()** System Call Example (Sheet 3 of 4)

```
        case 4:
            printf("\nEnter msq_bytes = ");
            scanf("%d", &bytes);
            buf->msg_qbytes = bytes;
            printf("\nmsg_qbytes = %d\n",
                buf->msg_qbytes);
            break;
        }

        /*Do the change.*/
        rtrn = msgctl(msqid, IPC_SET,
            buf);
        break;

    case 3:    /*Remove the msqid along with its
                  associated message queue
                  and data structure.*/
        rtrn = msgctl(msqid, IPC_RMID, NULL);
    }
    /*Perform the following if the call is unsuccessful.*/
    if(rtrn == -1)
    {
        printf ("\nThe msgctl system call failed!\n");
        printf ("The error number = %d\n", errno);
    }
    /*Return the msqid upon successful completion.*/
    else
        printf ("\nMsgctl was successful for msqid = %d\n",
            msqid);
    exit (0);
}
```

Figure 8-5: **msgctl()** System Call Example (Sheet 4 of 4)

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 *IRIS-4D Programmer's 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.

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 *IRIS-4D Programmer's 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 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 **msgbuf**1 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 workstation 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 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:

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()** 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**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the message operations, msgop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

struct msgbuf1 {
    long    mtype;
    char    mtext[8192];
} sndbuf, rcvbuf, *msgp;

/*Start of main C language program*/
main()
{
    extern int errno;
    int i, c, flag, flags, choice;
    int rtrn, msqid, msgsz, msgflg;
    long mtype, msgtyp;
    struct msqid_ds msqid_ds, *buf;
    buf = &msqid_ds;
```

Figure 8-6: **msgop()** System Call Example (Sheet 1 of 7)

```
/*Select the desired operation.*/
printf("Enter the corresponding\n");
printf("code to send or\n");
printf("receive a message:\n");
printf("Send            =  1\n");
printf("Receive         =  2\n");
printf("Entry           =  ");
scanf("%d", &choice);

if(choice == 1) /*Send a message.*/
{
    msgp = &sndbuf; /*Point to user send structure.*/

    printf("\nEnter the msqid of\n");
    printf("the message queue to\n");
    printf("handle the message = ");
    scanf("%d", &msqid);

    /*Set the message type.*/
    printf("\nEnter a positive integer\n");
    printf("message type (long) for the\n");
    printf("message = ");
    scanf("%d", &msgtyp);
    msgp->mtype = msgtyp;

    /*Enter the message to send.*/
    printf("\nEnter a message: \n");

    /*A control-d (^d) terminates as
      EOF.*/
```

Figure 8-6: **msgop()** System Call Example (Sheet 2 of 7)

```
/*Get each character of the message
  and put it in the mtext array.*/
for(i = 0; ((c = getchar()) != EOF); i++)
    sndbuf.mtext[i] = c;

/*Determine the message size.*/
msgsz = i + 1;

/*Echo the message to send.*/
for(i = 0; i < msgsz; i++)
    putchar(sndbuf.mtext[i]);

/*Set the IPC_NOWAIT flag if
  desired.*/
printf("\nEnter a 1 if you want the\n");
printf("the IPC_NOWAIT flag set:  ");
scanf("%d", &flag);
if(flag == 1)
    msgflg |= IPC_NOWAIT;
else
    msgflg = 0;

/*Check the msgflg.*/
printf("\nmsgflg = 0%o\n", msgflg);

/*Send the message.*/
rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
if(rtrn == -1)
printf("\nMsgsnd failed.  Error = %d\n",
        errno);
else {
    /*Print the value of test which
          should be zero for successful.*/
    printf("\nValue returned = %d\n", rtrn);
```

Figure 8-6: **msgop()** System Call Example (Sheet 3 of 7)

```
    /*Print the size of the message
     sent.*/
    printf("\nMsgsz = %d\n", msgsz);

    /*Check the data structure update.*/
    msgctl(msqid, IPC_STAT, buf);

    /*Print out the affected members.*/

    /*Print the incremented number of
     messages on the queue.*/
    printf("\nThe msg_qnum = %d\n",
      buf->msg_qnum);
    /*Print the process id of the last sender.*/
    printf("The msg_lspid = %d\n",
      buf->msg_lspid);
    /*Print the last send time.*/
    printf("The msg_stime = %d\n",
      buf->msg_stime);
  }
}

if(choice == 2)  /*Receive a message.*/
{
  /*Initialize the message pointer
   to the receive buffer.*/
  msgp = &rcvbuf;

  /*Specify the message queue which contains
     the desired message.*/
  printf("\nEnter the msqid = ");
  scanf("%d", &msqid);
```

Figure 8-6: **msgop()** System Call Example (Sheet 4 of 7)

```
/*Specify the specific message on the queue
      by using its type.*/
printf("\nEnter the msgtyp = ");
scanf("%d", &msgtyp);

/*Configure the control flags for the
      desired actions.*/
printf("\nEnter the corresponding code\n");
printf("to select the desired flags: \n");
printf("No flags                    =  0\n");
printf("MSG_NOERROR                 =  1\n");
printf("IPC_NOWAIT                  =  2\n");
printf("MSG_NOERROR and IPC_NOWAIT  =  3\n");
printf("                Flags       =  ");
scanf("%d", &flags);

switch(flags) {
    /*Set msgflg by ORing it with the appropriate
              flags (constants).*/
case 0:
    msgflg = 0;
    break;
case 1:
    msgflg |= MSG_NOERROR;
    break;
case 2:
    msgflg |= IPC_NOWAIT;
    break;
case 3:
    msgflg |= MSG_NOERROR | IPC_NOWAIT;
    break;
}
```

Figure 8-6: **msgop()** System Call Example (Sheet 5 of 7)

```
        /*Specify the number of bytes to receive.*/
        printf("\nEnter the number of bytes\n");
        printf("to receive (msgsz) = ");
        scanf("%d", &msgsz);

        /*Check the values for the arguments.*/
        printf("\nmsqid =%d\n", msqid);
        printf("\nmsgtyp = %d\n", msgtyp);
        printf("\nmsgsz = %d\n", msgsz);
        printf("\nmsgflg = 0%o\n", msgflg);

        /*Call msgrcv to receive the message.*/
        rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

        if(rtrn == -1)  {
            printf("\nMsgrcv failed.  ");
            printf("Error = %d\n", errno);
        }
        else {
            printf ("\nMsgctl was successful\n");
            printf("for msqid = %d\n",
                msqid);

            /*Print the number of bytes received,
              it is equal to the return
              value.*/
            printf("Bytes received = %d\n", rtrn);
```

Figure 8-6: **msgop()** System Call Example (Sheet 6 of 7)

```
                /*Print the received message.*/
                for(i = 0; i<=rtrn; i++)
                    putchar(rcvbuf.mtext[i]);
            }
            /*Check the associated data structure.*/
            msgctl(msqid, IPC_STAT, buf);
            /*Print the decremented number of messages.*/
            printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
            /*Print the process id of the last receiver.*/
            printf("The msg_lrpid = %d\n", buf->msg_lrpid);
            /*Print the last message receive time*/
            printf("The msg_rtime = %d\n", buf->msg_rtime);
        }
    }
```

Figure 8-6: **msgop()** System Call Example (Sheet 7 of 7)

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**(), semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl**() 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 *IRIS-4D Programmer's 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.

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 "nonblocking 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.

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:

- 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**()] 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**(). 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

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 *IRIS-4D Programmer's 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

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
CW| ORed 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

The **semflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));

semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **semget**(2) entry in the *IRIS-4D Programmer's 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:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

or

```
semid = semget ( 0, nsems, semflg);
```

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.

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:

```
      semflg = 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 *IRIS-4D Programmer's 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**.

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**.

```
/*This is a program to illustrate
**the semaphore get, semget(),
**system call capabilities.*/

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <errno.h>

/*Start of main C language program*/
main()
{
    key_t key;      /*declare as long integer*/
    int opperm, flags, nsems;
    int semid, opperm_flags;

    /*Enter the desired key*/
    printf("\nEnter the desired key in hex = ");
    scanf("%x", &key);

    /*Enter the desired octal operation
          permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);
```

Figure 8-9: **semget()** System Call Example (Sheet 1 of 3)

```
/*Set the desired flags.*/
printf("\nEnter corresponding number to\n");
printf("set the desired flags:\n");
printf("No flags                 = 0\n");
printf("IPC_CREAT                = 1\n");
printf("IPC_EXCL                 = 2\n");
printf("IPC_CREAT and IPC_EXCL    = 3\n");
printf("            Flags         = ");
/*Get the flags to be set.*/
scanf("%d", &flags);

/*Error checking (debugging)*/
printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
    key, opperm, flags);
/*Incorporate the control fields (flags) with
      the operation permissions.*/
switch (flags)
{
case 0:    /*No flags are to be set.*/
    opperm_flags = (opperm | 0);
    break;
case 1:    /*Set the IPC_CREAT flag.*/
    opperm_flags = (opperm | IPC_CREAT);
    break;
case 2:    /*Set the IPC_EXCL flag.*/
    opperm_flags = (opperm | IPC_EXCL);
    break;
case 3: /*Set the IPC_CREAT and IPC_EXCL
              flags.*/
    opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
}
```

Figure 8-9: **semget()** System Call Example (Sheet 2 of 3)

```
        /*Get the number of semaphores for this set.*/
        printf("\nEnter the number of\n");
        printf("desired semaphores for\n");
        printf("this set (25 max) = ");
        scanf("%d", &nsems);

        /*Check the entry.*/
        printf("\nNsems = %d\n", nsems);

        /*Call the semget system call.*/
        semid = semget(key, nsems, opperm_flags);

        /*Perform the following if the call is unsuccessful.*/
        if(semid == -1)
        {
            printf("The semget system call failed!\n");
            printf("The error number = %d\n", errno);
        }
        /*Return the semid upon successful completion.*/
        else
            printf("\nThe semid = %d\n", semid);
        exit(0);
    }
```

Figure 8-9: **semget**() System Call Example (Sheet 3 of 3)

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 *IRIS-4D Programmer's 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
- 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 *IRIS-4D Programmer's 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
- **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.

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 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**.

```
/*This is a program to illustrate
**the semaphore control, semctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    struct semid_ds semid_ds;
    int c, i, length;
    int uid, gid, mode;
    int retrn, semid, semnum, cmd, choice;
    union semun  {
        int val;
        struct semid_ds *buf;
        ushort array[25];
    } arg;

    /*Initialize the data structure pointer.*/
    arg.buf = &semid_ds;
```

Figure 8-10: **semctl()** System Call Example (Sheet 1 of 7)

```
        /*Enter the semaphore ID.*/
        printf("Enter the semid = ");
        scanf("%d", &semid);

        /*Choose the desired command.*/
        printf("\nEnter the number for\n");
        printf("the desired cmd:\n");
        printf("GETVAL      =  1\n");
        printf("SETVAL      =  2\n");
        printf("GETPID      =  3\n");
        printf("GETNCNT     =  4\n");
        printf("GETZCNT     =  5\n");
        printf("GETALL      =  6\n");
        printf("SETALL      =  7\n");
        printf("IPC_STAT    =  8\n");
        printf("IPC_SET     =  9\n");
        printf("IPC_RMID    =  10\n");
        printf("Entry       =  ");
        scanf("%d", &cmd);

        /*Check entries.*/
        printf ("\nsemid =%d, cmd = %d\n\n",
            semid, cmd);

        /*Set the command and do the call.*/
        switch (cmd)
        {
```

Figure 8-10: **semctl()** System Call Example (Sheet 2 of 7)

```
case 1: /*Get a specified value.*/
    printf("\nEnter the semnum = ");
    scanf("%d", &semnum);
    /*Do the system call.*/
    retrn = semctl(semid, semnum, GETVAL, 0);
    printf("\nThe semval = %d\n", retrn);
    break;
case 2: /*Set a specified value.*/
    printf("\nEnter the semnum = ");
    scanf("%d", &semnum);
    printf("\nEnter the value = ");
    scanf("%d", &arg.val);
    /*Do the system call.*/
    retrn = semctl(semid, semnum, SETVAL, arg.val);
    break;
case 3: /*Get the process ID.*/
    retrn = semctl(semid, 0, GETPID, 0);
    printf("\nThe sempid = %d\n", retrn);
    break;
case 4: /*Get the number of processes
      waiting for the semaphore to
      become greater than its current
              value.*/
    printf("\nEnter the semnum = ");
    scanf("%d", &semnum);
    /*Do the system call.*/
    retrn = semctl(semid, semnum, GETNCNT, 0);
    printf("\nThe semncnt = %d", retrn);
    break;
```

Figure 8-10: **semctl**() System Call Example (Sheet 3 of 7)

```
        case 5: /*Get the number of processes
              waiting for the semaphore
                      value to become zero.*/
            printf("\nEnter the semnum = ");
            scanf("%d", &semnum);
            /*Do the system call.*/
            retrn = semctl(semid, semnum, GETZCNT, 0);
            printf("\nThe semzcnt = %d", retrn);
            break;

        case 6: /*Get all of the semaphores.*/
            /*Get the number of semaphores in
              the semaphore set.*/
            retrn = semctl(semid, 0, IPC_STAT, arg.buf);
            length = arg.buf->sem_nsems;
            if(retrn == -1)
                goto ERROR;
            /*Get and print all semaphores in the
              specified set.*/
            retrn = semctl(semid, 0, GETALL, arg.array);
            for (i = 0; i < length; i++)
            {
                printf("%d", arg.array[i]);
                /*Seperate each
                  semaphore.*/
                printf("%c", ' ');
            }
            break;
```

Figure 8-10: **semctl()** System Call Example (Sheet 4 of 7)

```
    case 7: /*Set all semaphores in the set.*/
        /*Get the number of semaphores in
          the set.*/
        retrn = semctl(semid, 0, IPC_STAT, arg.buf);
        length = arg.buf->sem_nsems;
        printf("Length = %d\n", length);
        if(retrn == -1)
            goto ERROR;
        /*Set the semaphore set values.*/
        printf("\nEnter each value:\n");
        for(i = 0; i < length ; i++)
        {
            scanf("%d", &c);
            arg.array[i] = c;
        }
        /*Do the system call.*/
        retrn = semctl(semid, 0, SETALL, arg.array);
        break;

    case 8: /*Get the status for the semaphore set.*/
        /*Get and print the current status values.*/
        retrn = semctl(semid, 0, IPC_STAT, arg.buf);
        printf ("\nThe USER ID = %d\n",
            arg.buf->sem_perm.uid);
        printf ("The GROUP ID = %d\n",
            arg.buf->sem_perm.gid);
        printf ("The operation permissions = 0%o\n",
            arg.buf->sem_perm.mode);
        printf ("The number of semaphores in set = %d\n",
            arg.buf->sem_nsems);
        printf ("The last semop time = %d\n",
            arg.buf->sem_otime);
```

Figure 8-10: **semctl()** System Call Example (Sheet 5 of 7)

```
        printf ("The last change time  = %d\n",
            arg.buf->sem_ctime);
        break;

    case 9:    /*Select and change the desired
                   member of the data structure.*/
        /*Get the current status values.*/
        retrn = semctl(semid, 0, IPC_STAT, arg.buf);
        if(retrn == -1)
            goto ERROR;
        /*Select the member to change.*/
        printf("\nEnter the number for the\n");
        printf("member to be changed:\n");
        printf("sem_perm.uid    = 1\n");
        printf("sem_perm.gid    = 2\n");
        printf("sem_perm.mode   = 3\n");
        printf("Entry           = ");
        scanf("%d", &choice);
        switch(choice){

        case 1: /*Change the user ID.*/
            printf("\nEnter USER ID = ");
            scanf ("%d", &uid);
            arg.buf->sem_perm.uid = uid;
            printf("\nUSER ID = %d\n",
                arg.buf->sem_perm.uid);
            break;

        case 2: /*Change the group ID.*/
            printf("\nEnter GROUP ID = ");
            scanf("%d", &gid);
            arg.buf->sem_perm.gid = gid;
            printf("\nGROUP ID = %d\n",
                arg.buf->sem_perm.gid);
            break;
```

Figure 8-10: **semctl()** System Call Example (Sheet 6 of 7)

```
        case 3: /*Change the mode portion of
              the operation
                          permissions.*/
            printf("\nEnter MODE = ");
            scanf("%o", &mode);
            arg.buf->sem_perm.mode = mode;
            printf("\nMODE = 0%o\n",
                arg.buf->sem_perm.mode);
            break;
        }
        /*Do the change.*/
        retrn = semctl(semid, 0, IPC_SET, arg.buf);
        break;
    case 10:    /*Remove the semid along with its
                  data structure.*/
        retrn = semctl(semid, 0, IPC_RMID, 0);
    }
    /*Perform the following if the call is unsuccessful.*/
    if(retrn == -1)
    {
ERROR:
        printf ("\n\nThe semctl system call failed!\n");
        printf ("The error number = %d\n", errno);
        exit(0);
    }
    printf ("\n\nThe semctl system call was successful\n");
    printf ("for semid = %d\n", semid);
    exit (0);
}
```

Figure 8-10: **semctl**() System Call Example (Sheet 7 of 7)

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 *IRIS-4D Programmer's 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. Sem**buf** 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
- 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 8-11) 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 *IRIS-4D Programmer's 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).

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). Sem**buf** 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**() 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**.

```
/*This is a program to illustrate
**the semaphore operations, semop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
/*Start of main C language program*/
main()
{
    extern int errno;
    struct sembuf sembuf[10], *sops;
    char string[];
    int retrn, flags, sem_num, i, semid;
    unsigned nsops;
    sops = sembuf; /*Pointer to array sembuf.*/

    /*Enter the semaphore ID.*/
    printf("\nEnter the semid of\n");
    printf("the semaphore set to\n");
    printf("be operated on = ");
    scanf("%d", &semid);
    printf("\nsemid = %d", semid);
```

Figure 8-11: **semop**(2) System Call Example (Sheet 1 of 4)

```
/*Enter the number of operations.*/
printf("\nEnter the number of semaphore\n");
printf("operations for this set = ");
scanf("%d", &nsops);
printf("\nnosops = %d", nsops);

/*Initialize the array for the
  number of operations to be performed.*/
for(i = 0; i < nsops; i++, sops++)
{

    /*This determines the semaphore in
      the semaphore set.*/
    printf("\nEnter the semaphore\n");
    printf("number (sem_num) = ");
    scanf("%d", &sem_num);
    sops->sem_num = sem_num;
    printf("\nThe sem_num = %d", sops->sem_num);

    /*Enter a (-)number to decrement,
      an unsigned number (no +) to increment,
      or zero to test for zero.  These values
      are entered into a string and converted
      to integer values.*/
    printf("\nEnter the operation for\n");
    printf("the semaphore (sem_op) = ");
    scanf("%s", string);
    sops->sem_op = atoi(string);
    printf("\nsem_op = %d\n", sops->sem_op);
```

Figure 8-11: **semop**(2) System Call Example (Sheet 2 of 4)

```
        /*Specify the desired flags.*/
        printf("\nEnter the corresponding\n");
        printf("number for the desired\n");
        printf("flags:\n");
        printf("No flags                  = 0\n");
        printf("IPC_NOWAIT                = 1\n");
        printf("SEM_UNDO                  = 2\n");
        printf("IPC_NOWAIT and SEM_UNDO   = 3\n");
        printf("            Flags         = ");
        scanf("%d", &flags);

        switch(flags)
        {
        case 0:
            sops->sem_flg = 0;
            break;
        case 1:
            sops->sem_flg = IPC_NOWAIT;
            break;
        case 2:
            sops->sem_flg = SEM_UNDO;
            break;
        case 3:
            sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
            break;
        }
        printf("\nFlags = 0%o\n", sops->sem_flg);
    }
```

Figure 8-11: **semop**(2) System Call Example (Sheet 3 of 4)

```
    /*Print out each structure in the array.*/
    for(i = 0; i < nsops; i++)
    {
        printf("\nsem_num = %d\n", sembuf[i].sem_num);
        printf("sem_op = %d\n", sembuf[i].sem_op);
        printf("sem_flg = %o\n", sembuf[i].sem_flg);
        printf("%c", ' ');
    }

    sops = sembuf; /*Reset the pointer to
                     sembuf[0].*/

    /*Do the semop system call.*/
    retrn = semop(semid, sops, nsops);
    if(retrn == -1)  {
        printf("\nSemop failed.  ");
        printf("Error = %d\n", errno);
    }
    else {
        printf ("\nSemop was successful\n");
        printf("for semid = %d\n", semid);

        printf("Value returned = %d\n", retrn);
    }
}
```

Figure 8-11: **semop(2)** System Call Example (Sheet 4 of 4)

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 using the **shmctl**(2) system call.

System calls, which are documented in the *IRIS-4D Programmer's 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 shmid_ds {
    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."

Figure 8-12 is a table that shows the shared memory state information.

Shared Memory States

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.

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 *IRIS-4D Programmer's 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.

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
| ORed 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:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));

shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **shmget**(2) entry in the *IRIS-4D Programmer's 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:

```
shmid = shmget (IPC_PRIVATE, size, shmflg);
```

or

```
shmid = shmget ( 0 , size, shmflg);
```

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 (**shmids**) 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:

```
      shmflg = 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 **shmid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique **shmid** is returned if the system call is successful. Any value for **shmflg** returns a new **shmid** 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.

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 *IRIS-4D Programmer's 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**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the shared memory get, shmget(),
**system call capabilities.*/

#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>
#include    <errno.h>

/*Start of main C language program*/
main()
{
    key_t key;              /*declare as long integer*/
    int opperm, flags;
    int shmid, size, opperm_flags;
    /*Enter the desired key*/
    printf("Enter the desired key in hex = ");
    scanf("%x", &key);

    /*Enter the desired octal operation
      permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);
```

Figure 8-15: **shmget**(2) System Call Example (Sheet 1 of 3)

```
/*Set the desired flags.*/
printf("\nEnter corresponding number to\n");
printf("set the desired flags:\n");
printf("No flags                  = 0\n");
printf("IPC_CREAT                 = 1\n");
printf("IPC_EXCL                  = 2\n");
printf("IPC_CREAT and IPC_EXCL    = 3\n");
printf("            Flags         = ");
/*Get the flag(s) to be set.*/
scanf("%d", &flags);

/*Check the values.*/
printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
    key, opperm, flags);

/*Incorporate the control fields (flags) with
  the operation permissions*/
switch (flags)
{
case 0:    /*No flags are to be set.*/
    opperm_flags = (opperm | 0);
    break;
case 1:    /*Set the IPC_CREAT flag.*/
    opperm_flags = (opperm | IPC_CREAT);
    break;
case 2:    /*Set the IPC_EXCL flag.*/
    opperm_flags = (opperm | IPC_EXCL);
    break;
case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
    opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
}
```

Figure 8-15: **shmget**(2) System Call Example (Sheet 2 of 3)

```
    /*Get the size of the segment in bytes.*/
    printf ("\nEnter the segment");
    printf ("\nsize in bytes = ");
    scanf ("%d", &size);

    /*Call the shmget system call.*/
    shmid = shmget (key, size, opperm_flags);

    /*Perform the following if the call is unsuccessful.*/
    if(shmid == -1)
    {
        printf ("\nThe shmget system call failed!\n");
        printf ("The error number = %d\n", errno);
    }
    /*Return the shmid upon successful completion.*/
    else
        printf ("\nThe shmid = %d\n", shmid);
    exit(0);
}
```

Figure 8-15: **shmget**(2) System Call Example (Sheet 3 of 3)

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 *IRIS-4D Programmer's Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_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 **shmid** 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 **shmid** and place it in the data structure pointed to by the ***buf** pointer in the user memory area
- 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 *IRIS-4D Programmer's 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.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **shmid_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 shared memory segment identifier which is stored at the address of the **shmid** 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**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the shared memory control, shmctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    int uid, gid, mode;
    int rtrn, shmid, command, choice;
    struct shmid_ds shmid_ds, *buf;
    buf = &shmid_ds;

    /*Get the shmid, and command.*/
    printf("Enter the shmid = ");
    scanf("%d", &shmid);
    printf("\nEnter the number for\n");
    printf("the desired command:\n");
```

Figure 8-16: **shmctl**(2) System Call Example (Sheet 1 of 6)

```
printf("IPC_STAT      =  1\n");
printf("IPC_SET       =  2\n");
printf("IPC_RMID      =  3\n");
printf("SHM_LOCK      =  4\n");
printf("SHM_UNLOCK    =  5\n");
printf("Entry         =  ");
scanf("%d", &command);

/*Check the values.*/
printf ("\nshmid =%d, command = %d\n",
    shmid, command);

switch (command)
{
case 1:    /*Use shmctl() to duplicate
      the data structure for
              shmid in the shmid_ds area pointed
              to by buf and then print it out.*/
    rtrn = shmctl(shmid, IPC_STAT,
        buf);
    printf ("\nThe USER ID = %d\n",
        buf->shm_perm.uid);
    printf ("The GROUP ID = %d\n",
        buf->shm_perm.gid);
    printf ("The creator's ID = %d\n",
        buf->shm_perm.cuid);
    printf ("The creator's group ID = %d\n",
        buf->shm_perm.cgid);
    printf ("The operation permissions = 0%o\n",
        buf->shm_perm.mode);
    printf ("The slot usage sequence\n");
```

Figure 8-16: **shmctl**(2) System Call Example (Sheet 2 of 6)

```
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 */
```

Figure 8-16: **shmctl**(2) System Call Example (Sheet 3 of 6)

```
case 2:    /*Select and change the desired
              member(s) of the data structure.*/

    /*Get the original data for this shmid
          data structure first.*/
    rtrn = shmctl(shmid, IPC_STAT, buf);

    printf("\nEnter the number for the\n");
    printf("member to be changed:\n");
    printf("shm_perm.uid   = 1\n");
    printf("shm_perm.gid   = 2\n");
    printf("shm_perm.mode  = 3\n");
    printf("Entry          = ");
    scanf("%d", &choice);
    /*Only one choice is allowed per
      pass as an illegal entry will
          cause repetitive failures until
      shmid_ds is updated with
          IPC_STAT.*/
```

Figure 8-16: **shmctl**(2) System Call Example (Sheet 4 of 6)

```
switch(choice){
case 1:
    printf("\nEnter USER ID = ");
    scanf ("%d", &uid);
    buf->shm_perm.uid = uid;
    printf("\nUSER ID = %d\n",
        buf->shm_perm.uid);
    break;

case 2:
    printf("\nEnter GROUP ID = ");
    scanf("%d", &gid);
    buf->shm_perm.gid = gid;
    printf("\nGROUP ID = %d\n",
        buf->shm_perm.gid);
    break;

case 3:
    printf("\nEnter MODE = ");
    scanf("%o", &mode);
    buf->shm_perm.mode = mode;
    printf("\nMODE = 0%o\n",
        buf->shm_perm.mode);
    break;
}
/*Do the change.*/
rtrn = shmctl(shmid, IPC_SET,
    buf);
break;
```

Figure 8-16: **shmctl()** System Call Example (Sheet 5 of 6)

```
    case 3:    /*Remove the shmid along with its
                  associated
                  data structure.*/
        rtrn = shmctl(shmid, IPC_RMID, NULL);
        break;

    case 4: /*Lock the shared memory segment*/
        rtrn = shmctl(shmid, SHM_LOCK, NULL);
        break;
    case 5: /*Unlock the shared memory
                  segment.*/
        rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
        break;
    }
    /*Perform the following if the call is unsuccessful.*/
    if(rtrn == -1)
    {
        printf ("\nThe shmctl system call failed!\n");
        printf ("The error number = %d\n", errno);
    }
    /*Return the shmid upon successful completion.*/
    else
        printf ("\nShmctl was successful for shmid = %d\n",
            shmid);
    exit (0);
}
```

Figure 8-16: **shmctl**(2) System Call Example (Sheet 6 of 6)

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 *IRIS-4D Programmer's 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.

The shm**addr** 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:

```
0xc00c0000
0xc00e0000
0xc0100000
0xc0120000
```

Note that these addresses are in chunks of 20,000 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 *IRIS-4D Programmer's 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
- **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**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the shared memory operations, shmop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include     <stdio.h>
#include     <sys/types.h>
#include     <sys/ipc.h>
#include     <sys/shm.h>
/*Start of main C language program*/
main()
{
    extern int errno;
    int flags, addr, i, attach;
    int shmid, shmflg, retrn, detach;



    /*Loop for attachments by this process.*/
    printf("Enter the number of\n");
    printf("attachments for this\n");
    printf("process (1-4).\n");
    printf("       Attachments = ");

    scanf("%d", &attach);
    printf("Number of attaches = %d\n", attach);
```

Figure 8-17: **shmop()** System Call Example (Sheet 1 of 4)

```
for(i = 1; i <= attach; i++) {
    /*Enter the shared memory ID.*/
    printf("\nEnter the shmid of\n");
    printf("the shared memory segment to\n");
    printf("be operated on = ");
    scanf("%d", &shmid);
    printf("\nshmid = %d\n", shmid);

    /*Enter the value for shmaddr.*/
    printf("\nEnter the value for\n");
    printf("the shared memory address\n");
    printf("in hexadecimal:\n");
    printf("          Shmaddr = ");
    scanf("%x", &addr);
    printf("The desired address = 0x%x\n", addr);

    /*Specify the desired flags.*/
    printf("\nEnter the corresponding\n");
    printf("number for the desired\n");
    printf("flags:\n");
    printf("SHM_RND              = 1\n");
    printf("SHM_RDONLY           = 2\n");
    printf("SHM_RND and SHM_RDONLY = 3\n");
    printf("          Flags      = ");
    scanf("%d", &flags);
```

Figure 8-17: **shmop()** System Call Example (Sheet 2 of 4)

```
        switch(flags)
        {
        case 1:
            shmflg = SHM_RND;
            break;
        case 2:
            shmflg = SHM_RDONLY;
            break;
        case 3:
            shmflg = SHM_RND | SHM_RDONLY;
            break;
        }
        printf("\nFlags = 0%o\n", shmflg);

        /*Do the shmat system call.*/
        retrn = (int)shmat(shmid, addr, shmflg);
        if(retrn == -1)  {
            printf("\nShmat failed.  ");
            printf("Error = %d\n", errno);
        }
        else {
            printf ("\nShmat was successful\n");
            printf("for shmid = %d\n", shmid);
            printf("The address = 0x%x\n", retrn);
        }
    }

    /*Loop for detachments by this process.*/
    printf("Enter the number of\n");
    printf("detachments for this\n");
    printf("process (1-4).\n");
    printf("       Detachments = ");
```

Figure 8-17: **shmop()** System Call Example (Sheet 3 of 4)

```
    scanf("%d", &detach);
    printf("Number of attaches = %d\n", detach);
    for(i = 1; i <= detach; i++) {

        /*Enter the value for shmaddr.*/
        printf("\nEnter the value for\n");
        printf("the shared memory address\n");
        printf("in hexadecimal:\n");
        printf("            Shmaddr = ");
        scanf("%x", &addr);
        printf("The desired address = 0x%x\n", addr);

        /*Do the shmdt system call.*/
        retrn = (int)shmdt(addr);
        if(retrn == -1)  {
            printf("Error = %d\n", errno);
        }
        else {
            printf ("\nShmdt was successful\n");
            printf("for address  = 0%x\n", addr);

        }
    }
}
```

Figure 8-17: **shmop()** System Call Example (Sheet 4 of 4)

The Terminal Information Utilities Package

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 *IRIS-4D Programmer's 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.

- 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.

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.** 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.

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.

Here's a simple shell script that uses the **terminfo** database.

```
#   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 an IRIS monitor 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 screen has to print the `BullsEye` in the middle of it. The description of the IRIS monitor in the **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=5425
export TERM
tput init
```

The first line names the terminal type, and the second line exports it. (See **profile**(4) in the *IRIS-4D Programmer's Reference Manual.*) The third line of the example tells the UNIX system to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. (The order of these lines is important. **$TERM** must be defined and exported first, so that when **tput** is called the proper initialization for the current terminal takes place.) If you had these lines in your **.profile** and you ran a **curses** program, the program would get the information that it needs about your terminal from the file **/usr/lib/terminfo/a/att5425**, which provides a match for **$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:

captoinfo(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
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 **term**(5). For more information about any of these components, see the *IRIS-4D Programmer's Reference Manual* and the *IRIS-4D User's Reference Manual.*

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.

CAUTION

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(), 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

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 the Teletype 5425 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 the Teletype 5425 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 *IRIS-4D Programmer's 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**. 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
tput init
```

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.

If a **curses** program does not run as expected, you might want to debug it with **edge**(1), which is documented in the *IRIS-4D Programmer's Reference Manual*). When using **edge**, 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 **edge**, 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.

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 spreadsheet.

Figure 9-4 represents what a pad, a subwindow, and some other windows could look like in comparison to a terminal screen.

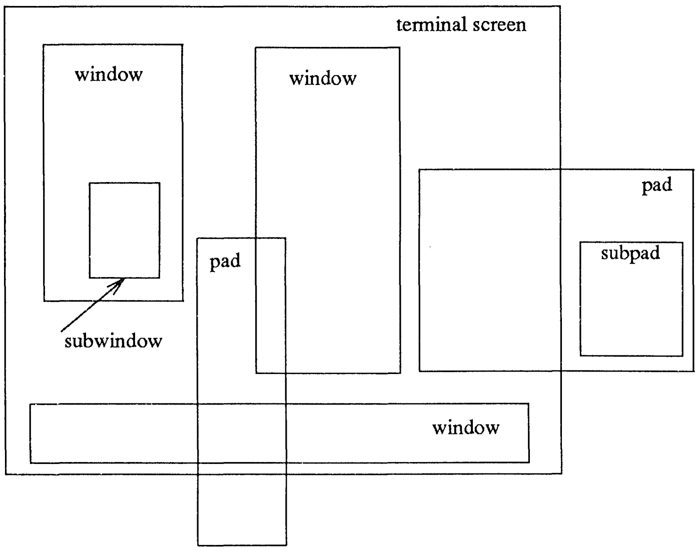


Figure 9-4: Multiple Windows and Pads Mapped to a Terminal Screen

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()**, **clrtobot()**

Following are descriptions and examples of these routines.

CAUTION 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.

addch()

SYNOPSIS

#include <curses.h>

int addch(ch)
chtype ch;

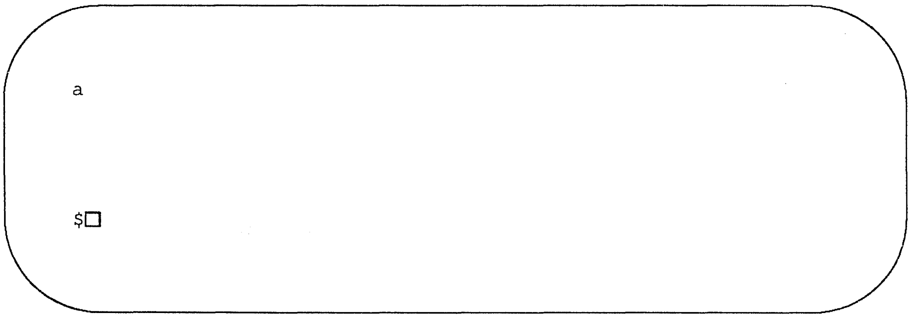
NOTES

- **addch()** writes a single character to **stdscr**.
- The character is of the type **chtype**, which is defined in **<curses.h>**. **chtype** 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 **chtype** and not as the basic type (for example, **short**) that **chtype** 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.

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.

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 and 9-2.

printw()

SYNOPSIS

#include <curses.h>

int printw(fmt [,arg...])
char *fmt

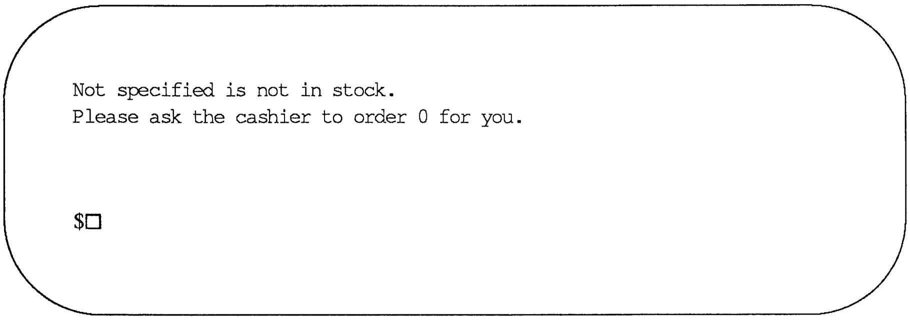
NOTES

- **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.



$□
```

move()

SYNOPSIS

#include <curses.h>

int move(y, x);
int y, x;

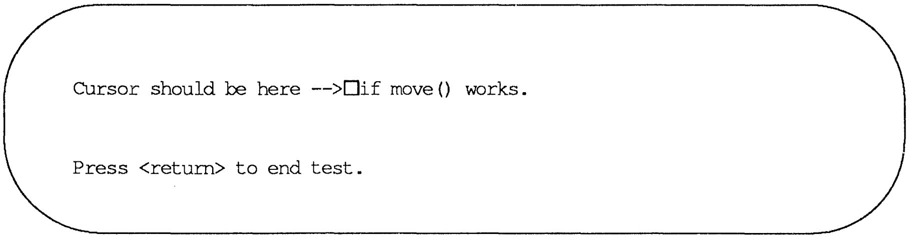
NOTES

- **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

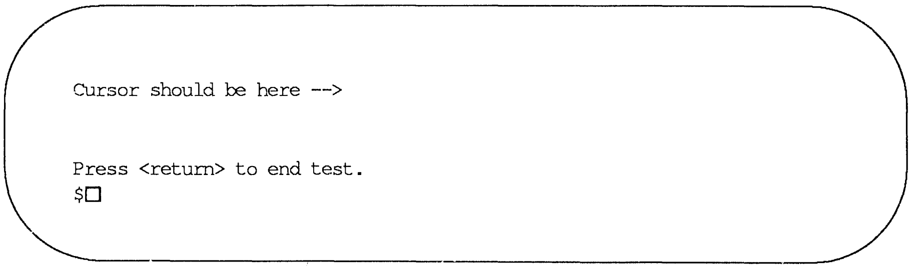
```
#include <curses.h>

main()
{
   initscr();
   addstr("Cursor should be here --> if move() works.");
   printw("\n\n\nPress <return> to end test.");
   move(0,25);
   refresh();
   getch();       /* Gets <return>; discussed below. */
   endwin();
}
```

Here's the output generated by running this program:



```
Cursor should be here -->□if move() works.


Press <return> to end test.
```

After you press **<return>**, the screen looks like this:



```
Cursor should be here -->


Press <return> to end test.
$□
```

See the **scatter** program under "**curses** Program Examples" in this chapter for another example of using **move()**.

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.

clrtoeol() and clrtobot()

SYNOPSIS

#include <curses.h>

int clrtoeol()
int clrtobot()

NOTES

- **clrtoeol**() changes the remainder of a line to all blanks.
- **clrtobot**() changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

EXAMPLE

The following sample program uses **clrtobot()**.

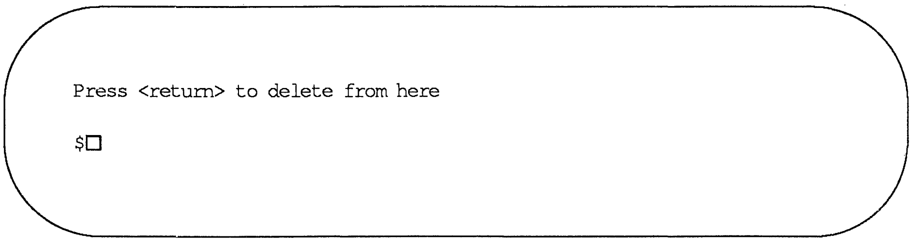
```
#include <curses.h>

main()
{
   initscr();
   addstr("Press <return> to delete from here to the end of the line and on.");
   addstr("\nDelete this too.\nAnd this.");
   move(0,30);
   refresh();
   getch();
   clrtobot();
   refresh();
   endwin();
}
```

Here's the output generated by running this program:

```
Press <return> 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 **<return>**:



```
Press <return> to delete from here

$□
```

See the **show** and **two** programs under "**curses** Example Programs" for examples of uses for **clrtoeol()**.

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.

getch()

SYNOPSIS

#include <curses.h>

int getch()

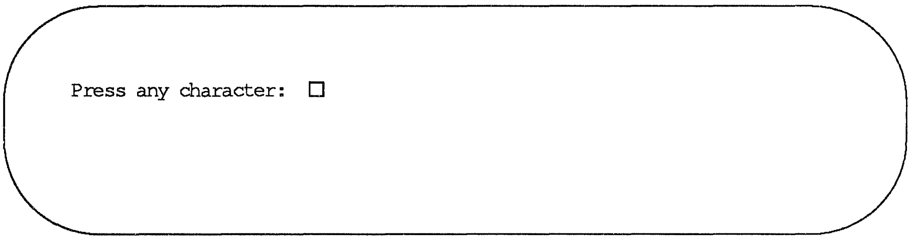
NOTES

- **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("\n\n\nThe character entered was a '%c'.\n", ch);
   refresh();
   endwin();
}
```

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:

```
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.

getstr()

SYNOPSIS

#include <curses.h>

int getstr(str)
char *str;

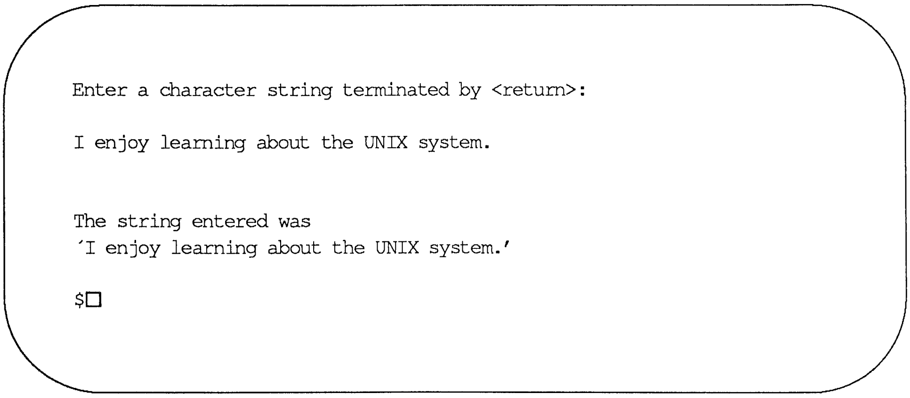
NOTES

- **getstr**() reads characters and stores them in a buffer until a **<return>**, **<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).

EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

   initscr();
   cbreak();       /* Explained later in the section "Input Options" */
   addstr("Enter a character string terminated by <return>:\n\n");
   refresh()
   getstr(str);
   printw("\n\n\nThe string entered was \n'%s'\n", str);
   refresh();
   endwin();
}
```

Assume you entered the string 'I enjoy learning about the UNIX system.' The final screen (after entering **<return>**) would appear as follows:



```
Enter a character string terminated by <return>:

I enjoy learning about the UNIX system.


The string entered was
'I enjoy learning about the UNIX system.'

$□
```

scanw()

SYNOPSIS

#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;

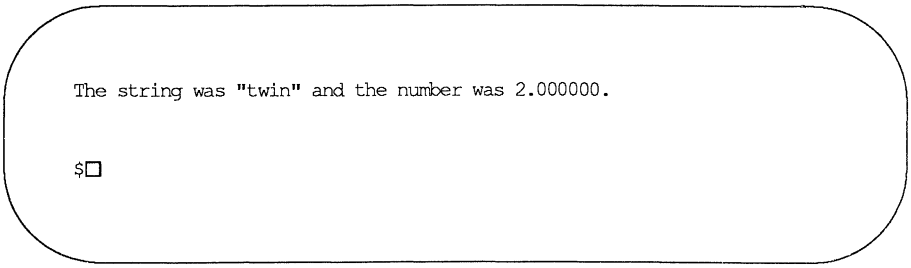
NOTES

- **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.

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()**.

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 of 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.

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.

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.)

beep() and flash()

SYNOPSIS

#include <curses.h>

int flash()
int beep()

NOTES

- **flash()** tries to flash the terminals 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 an erase character (typically #) and a line kill character (typically @)
- interprets a ctrl-D (control d) as end of file (EOF)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates **<return>** 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 9-5 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 9-5 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.

Input Options	**Characters**	
	Interpreted	**Uninterpreted**
Normal 'out of **curses** state'	interrupt, quit stripping **<return>** to <NL> echoing erase, kill EOF	
Normal **curses** '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()	**<return>** to <NL>	
nonl()		**<return>** to <NL>
raw() (instead of **cbreak()**)		break, quit stripping

Figure 9-5: Input Option Settings for **curses** Programs

CAUTION 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 9-5, 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.

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 9-5 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.

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 9-5 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

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 **wclrtobot**(*win*)
- **wrefresh**()

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the 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 **curses**(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 **curses** 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.

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.

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.

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.

CAUTION Subwindows of subwindows do not work (as of the copyright date of this *IRIS 4D Programmer's 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 ws 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.

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.

CAUTION 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 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 *IRIS-4D Programmer's 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_init()**, **slk_set()**, **slk_refresh()** and **slk_noutrefresh()**, **slk_clear**, and **slk_restore**.

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. 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.

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 9-6 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 9-6: 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.

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 9-7.

```
#include <curses.h>
#include <term.h>
...
   setupterm( (char*)0, 1, (int*)0 );
   ...
   putp(clear_screen);
   ...
   reset_shell_mode();
   exit(0);
```

Figure 9-7: 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.

```
/*
 * 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)
         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);
        }
    }

    /*
     * 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, outc***)** 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 *outc* 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.

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.

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.
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. (Lest we forget the UNIX motto: Build on the work of others.)

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 IRIS Series 5000 workstation:

```
5000|iris5000|IRIS Series 5000,
```

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 5000 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:

  stty –echo; cat –vu
  Type in the keys you want to test;
  for example, see what right arrow (→) transmits.
  <return>
  <ctrl-D>
  stty echo

  or

  cat >dev/null
  Type in the escape sequences you want to test;

for example, see what \E [**H** transmits.
<ctrl-D>

- The first line in each of these testing methods 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.
\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 **<return>** 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 (**cud1**).
- 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,
```

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:

```
mytermImytmImineIfancyIterminalIMy 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,
                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, from (0,0), the sequence

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,
        kcub1=\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 **/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<return>
(The trace information appears as the compilation
proceeds.)

Refer to the **tic**(1M) manual page in the *IRIS-4D System Administrator's 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 [–T*type*] *capname*

The type of terminal you are requesting information about is identified with the –T*type* 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 *IRIS-4D User's 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

CAUTION

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.

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.

```
 /* 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 <stdio.h>
#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();
        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()
{
        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;
```

```
/* 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;
                        }
                }
}

/*
 * 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();
}
```

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()**.

```
/*
 * 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)
        {
                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 ) {

                  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, which we have not previously discussed, is called to enable more cursor optimization. The **idlok()** routine, which we also have not discussed, is called to allow insert and delete line. (See **curses**(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 **curses()** 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);
        }
```

```
          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))
                              {
                                        clrtobot();
                                        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], "w+");
        signal(SIGINT, done);       /* die gracefully */

        me = newterm(getenv("TERM"), stdout, stdin); /* initialize my tty */
        you = newterm(argv[2], fdyou, fdyou);/* Initialize 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);
         for (line = 0; line < LINES - 1; line++) {
              if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
              clrtobot();
              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 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 **curses**(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 (;;)

        {
                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);
                }

        }
}
```

An Overview of the make Utility

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

make(1) provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways.

An individual programmer can easily forget

- file-to-file dependencies
- files that were modified and the impact that has on other files
- the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to

- find the target in the description file
- ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile**, **Makefile**, or **s.[mM]akefile**. If this naming convention is followed, the simple command **make** is usually sufficient to regenerate the target regardless of the number of files edited since the last **make**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file
- filenames and last-modified times from the file system
- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the **math** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lm  -o  prog

x.o  y.o :   defs.h
```

If this information were stored in a file named **makefile**, the command

```
make
```

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc –c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

make x.o

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions" below.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by what the macro stands for. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

$***, **$@, **$?**, and **$<** are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under "Description Files and Substitutions.") The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command

make LIBES="-ll -lm"

loads the three objects with both the **lex** (**-ll**) and the **math** (**-lm**) libraries, because macro definitions on the command line override definitions in the description file. (In UNIX system commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:   $(OBJECTS)
        $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        @size make

$(OBJECTS):  defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

                  # print files that are out-of-date
                  # with respect to "print" file.

print:   $(FILES)
         pr $? | $(LP)
         touch print
```

The **make** program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```
cc  -O -c main.c
cc  -O -c doname.c
cc  -O -c misc.c
cc  -O -c files.c
cc  -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -O -c gram.c
cc  main.o doname.o misc.o files.o dosys.o
     gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an at sign, @, in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is a sharp, #, and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp are totally ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (See Figure 10-2 at the end of the chapter.)

General Form

The general form of an entry in a description file is

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
 . . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp, #, except when the sharp is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries" section later in this chapter.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (–s option of the **make** command) or if the command line in the description file begins with an @ sign. **make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **–i** flag has been specified on the **make** command line, if the fake target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The $@ macro is set to the full target name of the current target. The $@ macro is evaluated only for explicitly named dependencies. The $? macro is set to the string of names that were found to be younger than the target. The $? macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the $< macro is the name of the related file that caused the action; and the $* macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name DEFAULT are used. If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)** (see below).

Extensions of $*, $@, and $<

The internally generated macros $*, $@, and $< are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single character macro. The **F** refers to the filename part of the single character macro. These additions are useful when building hierarchical **makefiles**. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a command can be

```
cd $(<D); $(MAKE) $(<F)
```

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script, which can handle all the C language programs (i.e., those files ending in **.c**). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

The Recursive Makefile

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line, the line is executed even if the **–n** flag is set. Since the **–n** flag is exported across invocations of **make** (through the MAKEFLAGS variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile**(s) describes a set of software subsystems. For testing purposes, **make –n ...** can be executed and everything that would have been done will be printed including output from lower level invocations of **make**.

Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **–r** flag is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **.r** file to a **.o** file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro $< is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependents are added to the usual list. A .SUFFIXES line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.c	C source file
.c~	SCCS C source file
.f	FORTRAN source file
.f~	SCCS FORTRAN source file
.s	Assembler source file
.s~	SCCS Assembler source file
.y	yacc source grammar
.y~	SCCS **yacc** source grammar
.l	**lex** source grammar
.l~	SCCS **ex** source grammar
.h	Header file
.h~	SCCS header file
.sh	Shell file
.sh~	SCCS shell file

Figure 10-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

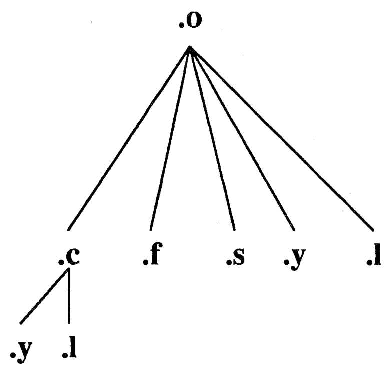


Figure 10-1: Summary of Default Transformation Path

If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 10-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, F77, YACC, and LEX. The command

make CC=newcc

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASFLAGS, CFLAGS, F77FLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus

make "CFLAGS=–g"

causes the **cc** command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library in the following manner:

```
projlib(object.o)
   or
projlib((entrypt))
```

where the second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object filename.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::   projlib(pfile1.o)
        $(CC) -c -O pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib::   projlib(pfile2.o)
        $(CC) -c -O pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

. . . and so on for each object . . .

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the filename being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde, ~, syntax will be described shortly.) The user may define other needed rules in the description file.

The above two-member library is then maintained with the following shorter **makefile:**

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
            @echo projlib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file; and the filename minus the suffix, respectively (**pfile1.c** and **pfile1**). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
projlib:    projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to **pfile1.c**. Also, there is no **pfile1.o** file.

1. **make projlib.**
2. Before **makeing projlib**, check each dependent of **projlib.**
3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.
4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)

5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.

6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **$@** (=**projlib**) and **$*** (=**pfile1**).

7. Look for a rule *X*.**a** and a file $*X*. The first *X* (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set **$<** to be **pfile1.c** and execute the rule. In fact, **make** must then compile **pfile1.c**.

8. The library has been updated. Execute the command associated with the **projlib:** dependency; namely

```
    @echo projlib up-to-date
```

It should be noted that to let **pfile1.o** have dependencies, the following syntax is required:

```
projlib(pfile1.o):         $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parenthesis.

SCCS Filenames: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the filename part of the complete pathname.

To allow **make** easy access to the prefix **s.** the tilde, ~, is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

.c~
.f~
.y~
.l~
.s~
.sh~
.h~

The following rules involving SCCS transformations are internally defined:

```
.c~:
.f~:
.sh~:
.c~.a:
.c~.c:
.c~.o:
.f~.a:
.f~.f:
.f~.o:
.s~.a:
.s~.s:
.s~.o:
.y~.c:
.y~.o:
.l~.l:
.l~.o:
.h~.h:
```

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS filename format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the UNIX system program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
        $(CC) $(CFLAGS) $< -o $@
```

In fact, this **.c:** rule is internally defined so no **makefile** is necessary at all. The user only needs to type

make cat dd echo date

(these are all UNIX system single-file programs) and all four C language source files are passed through the above shell command line associated with the **.c:** rule. The internally defined single suffix rules are:

.c:
.c~:
.f:
.f~:
.sh:
.sh~:

Others may be added in the **makefile** by the user.

include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** will read. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **includes** are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named **s.makefile** or **s.Makefile** exists, **make** will do a **get** on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The $$@ refers to the current "thing" to the left of the colon (which is $@). Also the form $$(@F) exists, which allows access to the file part of $@. Thus, in the following:

```
cat:     $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown

$(CMDS):          $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile**.

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make**(1) in the *IRIS-4D Programmer's Reference Manual.*

The make Command

The **make** command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

make [*options*] [*macro definitions*] [*targets*]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.

–t Touch the target files (causing them to be up to date) rather than issue the usual commands.

–q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

–p Print out the complete set of macro definitions and target descriptions.

–k Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.

–e Environment variables override assignments within **makefiles.**

–f Description filename. The next argument is assumed to be the name of a description file. A filename of – denotes the standard input. If there are no **–f** arguments, the file named **makefile** or **Makefile** or **s.[mM]akefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following two arguments are evaluated in the same manner as flags:

.DEFAULT	If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.
.PRECIOUS	Dependents on this target are not removed when quit or interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A macro, MAKEFLAGS, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the **makefile** update MAKEFLAGS. Thus, to describe how the environment interacts with **make**, the MAKEFLAGS macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **–f**, **–p**, and **–r** flags.)

2. Read the internal list of macro definitions.

3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).

4. Read the **makefile**(s). The assignments in the **makefile**(s) overrides the environment. This order is chosen so that when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the **–e** flag is used. The **–e** is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make –e ...** is typed, the variables in the environment override the definitions in the **makefile.** Also MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile.**

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions
2. environment
3. **makefile**(s)
4. command line

The **–e** flag has the effect of rearranging the order to:

1. internal definitions
2. **makefile**(s)
3. environment
4. command line

This order is general enough to allow a programmer to define a **makefile** or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a

```
#include "defs.h"
```

line, then the object file **x.o** depends on **defs.h**; the source file **x.c** does not. If **defs.h** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the **–n** option is very useful. The command

```
make –n
```

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a comment to an **include** file), the **–t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
make –ts
```

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **make** are reproduced below.

```
#
#       SUFFIXES RECOGNIZED BY MAKE
#

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~

#
#       PREDEFINED MACROS
#

MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
```

Figure 10-2: **make** Internal Rules (Sheet 1 of 5)

```
#
#        SINGLE SUFFIX RULES
#
.c:
         $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.c~:
         $(GET) $(GFLAGS) $<
         $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
         -rm -f $*.c
.f:
         $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@
.f~:
         $(GET) $(GFLAGS) $<
         $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
         -rm -f $*.f
.sh:
         cp $< $@; chmod 0777 $@
.sh~:
         $(GET) $(GFLAGS) $<
         cp $*.sh $*; chmod 0777 $@
         -rm -f $*.sh
```

Figure 10-2: **make** Internal Rules (Sheet 2 of 5)

```
#
#       DOUBLE SUFFIX RULES
#
.c~.c  .f~.f  .s~.s  .sh~.sh  .y~.y  .l~.l  .h~.h:
        $(GET) $(GFLAGS) $<

.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o

.c~.a:
        $(GET) $(GFLAGS) $<
        $(CC) -c $(CFLAGS) $*.c
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[co]

.c.o:
        $(CC) $(CFLAGS) -c $<

.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c

.f.a:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.o

.f~.a:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[fo]
```

Figure 10-2: **make** Internal Rules (Sheet 3 of 5)

```
.f.o:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
.f~.o:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        -rm -f $*.f
.s~.a:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[so]
.s.o:
        $(AS) $(ASFLAGS) -o $@ $<
.s~.o:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -rm -f $*.s
.l.c :
        $(LEX) $(LFLAGS) $<
        mv lex.yy.c $@
.l~.c:
        $(GET) $(GFLAGS) $<
        $(LEX) $(LFLAGS) $*.l
        mv lex.yy.c $@
```

Figure 10-2: **make** Internal Rules (Sheet 4 of 5)

```
.l.o:
         $(LEX) $(LFLAGS) $<
         $(CC) $(CFLAGS) -c lex.yy.c
         rm lex.yy.c
         mv lex.yy.o $@
         -rm -f $*.l
.l~.o:
         $(GET) $(GFLAGS) $<
         $(LEX) $(LFLAGS) $*.l
         $(CC) $(CFLAGS) -c lex.yy.c
         rm -f lex.yy.c $*.l
         mv lex.yy.o $*.o
.y.c :
         $(YACC) $(YFLAGS) $<
         mv y.tab.c $@
.y~.c :
         $(GET) $(GFLAGS) $<
         $(YACC) $(YFLAGS) $*.y
         mv y.tab.c $*.c
         -rm -f $*.y
.y.o:
         $(YACC) $(YFLAGS) $<
         $(CC) $(CFLAGS) -c y.tab.c
         rm y.tab.c
         mv y.tab.o $@
.y~.o:
         $(GET) $(GFLAGS) $<
         $(YACC) $(YFLAGS) $*.y
         $(CC) $(CFLAGS) -c y.tab.c
         rm -f y.tab.c $*.y
         mv y.tab.o $*.o
```

Figure 10-2: **make** Internal Rules (Sheet 5 of 5)

The Source Code Control System

The Source Code Control System (SCCS) is a maintenance and enhancement tracking tool that runs under the UNIX system. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version: why the changes were made, who made them, when they were made.

This guide covers the following:

- SCCS for Beginners: how to make, retrieve, and update an SCCS file
- Delta Numbering: how versions of an SCCS file are named
- SCCS Command Conventions: what rules apply to SCCS commands
- SCCS Commands: the fourteen SCCS commands and their more useful arguments
- SCCS Files: protection, format, and auditing of SCCS files

Neither the implementation of SCCS nor the installation procedure for SCCS is described in this guide.

SCCS for Beginners

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned an SID (SCCS Identification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file, that is, its first delta is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. More on delta numbering later. At this point, it is enough to know that by default SCCS assigns SIDs automatically.

Creating an SCCS File via admin

Suppose, for example, you have a file called **lang** that is simply a list of five programming language names. Use a text editor to create file **lang** containing the following list.

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of your **lang** file can be given to SCCS using the **admin** command (i.e., administer SCCS file). The following creates an SCCS file from the **lang** file:

```
admin –ilang s.lang
```

All SCCS files must have names that begin with **s.**, hence **s.lang**. The **–i** keyletter, together with its value **lang**, means **admin** is to create an SCCS file and initialize it with the contents of the file **lang**.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Ignore it for now. Its significance is described later with the **get** command under "SCCS Commands." In the following examples, this warning message is not shown although it may be issued.

Remove the **lang** file. It is no longer needed because it exists now under SCCS as **s.lang**.

```
rm lang
```

Retrieving a File via get

Use the **get** command as follows:

```
get s.lang
```

This retrieves **s.lang** and prints

```
1.1
5 lines
```

This tells you that **get** retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved text has been placed in a new file known as a "g.file." SCCS forms the g.file name by deleting the prefix **s.** from the name of the SCCS file. Thus, the original **lang** file has been recreated.

If you list, **ls**(1), the contents of your directory, you will see both **lang** and **s.lang**. SCCS retains **s.lang** for use by other users.

The **get s.lang** command creates **lang** as read-only and keeps no information regarding its creation. Because you are going to make changes to it, **get** must be informed of your intention to do so. This is done as follows:

```
get –e s.lang
```

get –e causes SCCS to create **lang** for both reading and writing (editing). It also places certain information about **lang** in another new file, called the "p.file" (**p.lang** in this case), which is needed later by the **delta** command.

get –e prints the same messages as **get**, except that now the SID for the first delta you will create is issued:

```
1.1
new delta  1.2
5 lines
```

Change **lang** by adding two more programming languages:

```
SNOBOL
ADA
```

Recording Changes via delta

Next, use the **delta** command as follows:

```
delta s.lang
```

delta then prompts with

```
comments?
```

Your response should be an explanation of why the changes were made. For example,

added more languages

delta now reads the p.file, **p.lang**, and determines what changes you made to **lang**. It does this by doing its own **get** to retrieve the original version and applying the **diff**(1) command to the original version and the edited version. Next, **delta** stores the changes in **s.lang** and destroys the no longer needed **p.lang** and **lang** files.

When this process is complete, **delta** outputs

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the delta you just created, and the next three lines summarize what was done to **s.lang**.

Additional Information about get

The command,

```
get s.lang
```

retrieves the latest version of the file **s.lang**, now 1.2. SCCS does this by starting with the original version of the file and applying the delta you made. If you use the get command now, any of the following will retrieve version 1.2.

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following **–r** are SIDs. When you omit the level number of the SID (as in **get –r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case also 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command.

```
get -e -r2 s.lang
```

Because release 2 does not exist, **get** retrieves the latest version before release 2. **get** also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output is

```
1.2
new delta 2.1
7 lines
```

which means version 1.2 has been retrieved, and 2.1 is the version **delta** will create. If the file is now edited, for example, by deleting COBOL from the list of languages, and **delta** is executed

```
delta s.lang
comments? deleted cobol from list of languages
```

you will see by **delta**'s output that version 2.1 is indeed created.

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release can be created in a similar manner.

The help Command

If the command

```
get lang
```

is now executed, the following message will be output:

```
ERROR [lang]: not an SCCS file (co1)
```

The code **co1** can be used with **help** to print a fuller explanation of the message.

help col

This gives the following explanation of why **get lang** produced an error message:

```
col:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

help is useful whenever there is doubt about the meaning of almost any SCCS message.

Delta Numbering

Think of deltas as the nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, etc. The components of these SIDs are called release and level numbers, respectively. Thus, normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Because the user may change the release number to indicate a major change, the release number then applies to all new deltas unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 11-1.

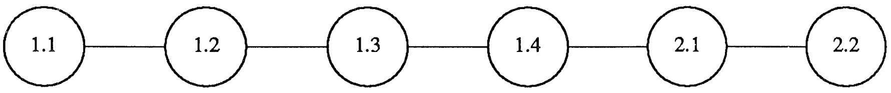


Figure 11-1: Evolution of an SCCS File

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned to a given delta that will not be dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 11-1. Assume that a production user reports a problem in version 1.3 that cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, etc. This is shown in Figure 11-2.

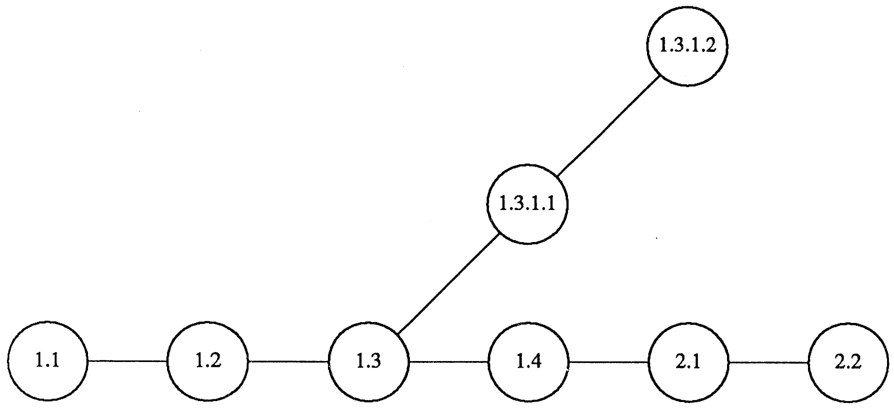


Figure 11-2: Tree Structure with Branch Deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 11-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, etc.), but the secondary branch number remains the same.

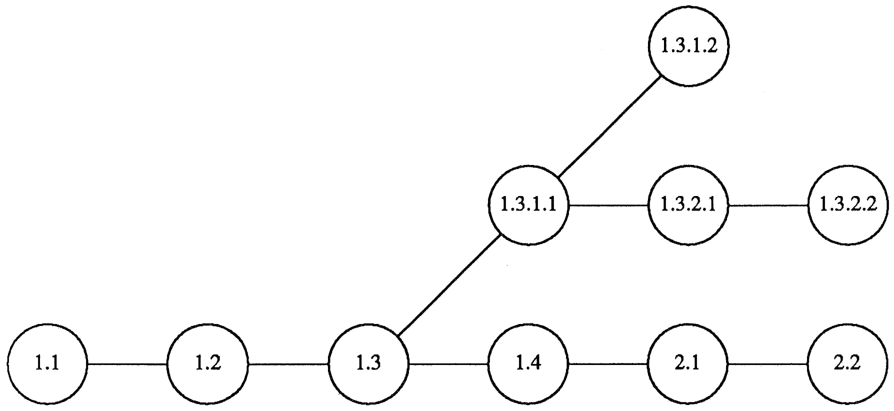


Figure 11-3: Extended Branching Concept

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

SCCS Command Conventions

SCCS commands accept two types of arguments:

- keyletters
- filenames

Keyletters are options that begin with a minus sign, –, followed by a lowercase letter and, in some cases, a value.

File and/or directory names specify the file(s) the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files (because of permission modes via **chmod**(1)) in the named directories are silently ignored.

In general, filename arguments may not begin with a minus sign. If a filename of – (a lone minus sign) is specified, the command will read the standard input (usually your terminal) for lines and take each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the commands **find**(1) or **ls**(1).

Keyletters are processed before filenames. Therefore, the placement of keyletters is arbitrary—that is, they may be interspersed with filenames. Filenames, however, are processed left to right. Somewhat different conventions apply to **help**(1), **what**(1), **sccsdiff**(1), and **val**(1), detailed later under "SCCS Commands."

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags will be discussed, but for a complete description see **admin**(1) in the *IRIS-4D Programmer's Reference Manual.*

The distinction between real user (see **passwd**(1)) and effective user will be of concern in discussing various actions of SCCS commands. For now, assume that the real and effective users are the same—the person logged into the UNIX system.

x.files and z.files

All SCCS commands that modify an SCCS file do so by writing a copy called the "x.file." This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. SCCS names the x.file by replacing the **s.** of the SCCS filename with **x.**. The x.file is created in the same directory as the SCCS file, given the same mode (see **chmod**(1)), and is owned by the effective user. When processing is complete, the old SCCS file is destroyed and the modified x.file is renamed (**x.** is relaced by **s.**) and becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the "z.file." SCCS forms its name by replacing the **s.** of the SCCS filename with a **z.** prefix. The z.file contains the process number of the command that creates it, and its existence prevents other commands from processing the SCCS file. The z.file is created with access permission mode 444 (read only) in the same directory as the SCCS file and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, users can ignore x.files and z.files. They are useful only in the event of system crashes or similar situations.

Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses can be used as an argument to the **help** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

SCCS Commands

This section describes the major features of the fourteen SCCS commands and their most common arguments. Full descriptions with details of all arguments are in the *IRIS-4D Programmer's Reference Manual.*

Here is a quick-reference overview of the commands:

get	retrieves versions of SCCS files
unget	undoes the effect of a **get –e** prior to the file being **delta**ed
delta	applies deltas (changes) to SCCS files and creates new versions
admin	initializes SCCS files, manipulates their descriptive text, and controls delta creation rights
prs	prints portions of an SCCS file in user specified format
sact	prints information about files that are currently out for edit
help	gives explanations of error messages
rmdel	removes a delta from an SCCS file allows removal of deltas created by mistake
cdc	changes the commentary associated with a delta
what	searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it useful in finding identifying information inserted by the **get** command
sccsdiff	shows differences between any two versions of an SCCS file
comb	combines consecutive deltas into one to reduce the size of an SCCS file
val	validates an SCCS file
vc	a filter that may be used for version control

The get Command

The **get**(1) command creates a file that contains a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the "g.file." It is created in the current directory and is owned by the real user. The mode assigned to the g.file depends on how the **get** command is used.

The most common use of **get** is

get s.abc

which normally retrieves the latest version of file **abc** from the SCCS file tree trunk and produces (for example) on the standard output

```
1.3
67 lines
No id keywords (cm7)
```

meaning version 1.3 of file **s.abc** was retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords were substituted in the file.

The generated g.file (file **abc**) is given access permission mode 444 (read only). This particular way of using **get** is intended to produce g.files only for inspection, compilation, etc. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example,

get s.abc s.xyz

produces

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.xyz:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a g.file for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, etc. within the g.file. This information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs, %. For example,

%I%

is the ID keyword replaced by the SID of the retrieved version of a file. Similarly, **%H%** and **%M%** are the names of the g.file. Thus, executing **get** on an SCCS file that contains the PL/I declaration,

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/18/85');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get** although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately twenty ID keywords provided, see **get**(1) in the *IRIS-4D Programmer's Reference Manual.*

Retrieval of Different Versions

The version of an SCCS file **get** retrieves is the most recently created delta of the highest numbered trunk release. However, any other version can be retrieved with **get –r** by specifying the version's SID. Thus,

```
get –r1.3 s.abc
```

retrieves version 1.3 of file **s.abc** and produces (for example) on the standard output

```
1.3
64 lines
```

A branch delta may be retrieved similarly,

```
get –r1.5.2.3 s.abc
```

which produces (for example) on the standard output

```
1.5.2.3
234 lines
```

When a SID is specified and the particular version does not exist in the SCCS file, an error message results.

Omitting the level number, as in

```
get –r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release. Thus, the above command might output,

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume release 9 does not exist in file **s.abc** and release 7 is the highest-numbered release below 9. Executing

```
get –r9 s.abc
```

might produce

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file **s.abc** below release 9. Similarly, omitting the sequence number, as in

```
get –r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

get –t will retrieve the latest (top) version of a particular release when no **–r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get –r3 –t s.abc
```

might produce

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce

```
3.2.1.5
46 lines
```

Retrieval With Intent to Make a Delta

get –e indicates an intent to make a delta. First, **get** checks the following.

1. The user list to determine if the login name or group ID of the person executing **get** is present. The login name or group ID must be present for the user to be allowed to make deltas. (See "The **admin** Command" for a discussion of making user lists.)
2. The release number (R) of the version being retrieved satisfies the relation

   floor is less than or equal to R, which is
   less than or equal to ceiling

   to determine if the release being accessed is a protected release. The floor and ceiling are flags in the SCCS file representing start and end of range.
3. The R is not locked against editing. The lock is a flag in the SCCS file.
4. Whether multiple concurrent edits are allowed for the SCCS file by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **get –e** causes the creation of a g.file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g.file already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a g.file being edited for the purpose of making a delta.

Any ID keywords appearing in the g.file are not substituted by **get –e** because the generated g.file is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. Because of this, **get** does not need to check for their presence in the g.file. Thus, the message

```
No id keywords (cm7)
```

is never output when **get –e** is used.

In addition, **get –e** causes the creation (or updating) of a p.file that is used to pass information to the **delta** command.

The following

```
get -e s.abc
```

produces (for example) on the standard output

```
1.3
new delta 1.4
67 lines
```

The unget Command

There may be times when a file is retrieved for editing in error; there is really no editing that needs to be done at this time. In such cases, the **unget** command can be used to cancel the delta reservation that was set up.

Additional get Options

If **get –r** and/or **–t** are used together with **–e**, the version retrieved for editing is the one specified with **–r** and/or **–t**.

get –i and **–x** are used to specify a list (see **get**(1) in the *IRIS-4D Programmer's Reference Manual* for the syntax of such a list) of deltas to be included and excluded, respectively. Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **get** checks for possible interference with other deltas. Two deltas can interfere, for example, when each one changes the same line of the retrieved g.file. A warning shows the range of lines within the retrieved g.file where the problem may exist. The user should examine the g.file to determine what the problem is and take appropriate corrective steps (e.g., edit the file).

CAUTION **get –i** and **get –x** should be used with extreme care.

get –k is used either to regenerate a g.file that may have been accidentally removed or ruined after **get –e**, or simply to generate a g.file in which the replacement of ID keywords has been suppressed. A g.file generated by **get –k** is identical to one produced by **get –e**, but no processing related to the p.file takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **get –e** commands may be executed on the same file as long as no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The p.file created by **get –e** is named by automatic replacement of the SCCS filename's prefix **s.** with **p.**. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p.file contains the following information for each delta that is still in progress:

- the SID of the retrieved version
- the SID given to the new delta when it is created
- the login name of the real user executing **get**

The first execution of **get –e** causes the creation of a p.file for the corresponding SCCS file. Subsequent executions only update the p.file with a line containing the above information. Before updating, however, **get** checks to assure that no entry already in the p.file specifies that the SID of the version to be retrieved is already retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress and processing continues. If the check fails, an error message results.

It should be noted that concurrent executions of **get** must be carried out from different directories. Subsequent executions from the same directory will attempt to overwrite the g.file, which is an SCCS error condition. In practice, this problem does not arise since each user normally has a different working directory. See "Protection" under "SCCS Files" for a discussion of how different users are permitted to use SCCS commands on the same files.

Figure 11-4 shows the possible SID components a user can specify with **get** (left-most column), the version that will then be retrieved by **get**, and the resulting SID for the delta, which **delta** will create (right-most column).

SID Specified in get*	–b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta To be Created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R< mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor number in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1

Figure 11-4: Determination of New SID (sheet 1 of 2)

SID Specified in get*	–b Key-Letter used†	Other Condition	SID Retrieved by get	SID of Delta to be Created by delta
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mS+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

Figure 11-4: Determination of New SID (sheet 2 of 2)

Footnotes to Figure 11-4:

* R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.

† The –**b** keyletter is effective only if the **b** flag (see **admin(1)**) is present in the file. An entry of – means irrelevant.

‡ This case applies if the **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as if specified on the command line. Thus, one of the other cases in this figure applies.

§ This is used to force the creation of the first delta in a new release.

** hR is the highest existing release that is lower than the specified, nonexistent release R.

Concurrent Edits of Same SID

Under normal conditions, more than one **get –e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get –e** is executed on the same SID.

Multiple concurrent edits are allowed if the **j** flag is set in the SCCS file. Thus:

```
get –e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by

```
get –e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening **delta**. In this case, a **delta** after the first **get** will produce delta 1.2 (assuming 1.1 is the most recent trunk delta), and a **delta** after the second **get** will produce delta 1.1.1.1.

Keyletters That Affect Output

get –p causes the retrieved text to be written to the standard output rather than to a g.file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. **get –p** is used, for example, to create a g.file with an arbitrary name, as in

get –p s.abc > *arbitrary-file-name*

get –s suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. **get –s** is used to prevent non-diagnostic messages from appearing on the user's terminal and is often used with **–p** to pipe the output, as in

```
get –p –s s.abc | pg
```

get –g suppresses the retrieval of the text of an SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file

```
get –g –r4.3 s.abc
```

outputs the SID 4.3 if it exists in the SCCS file **s.abc** or an error message if it does not. Another use of **get –g** is in regenerating a p.file that may have been accidentally destroyed, as in

```
get –e –g s.abc
```

get –l causes SCCS to create an "l.file." It is named by replacing the **s.** of the SCCS filename with **l.**, created in the current directory with mode 444 (read only) and owned by the real user. The l.file contains a table (whose format is described under **get**(1) in the *IRIS-4D Programmer's Reference Manual*) showing the deltas used in constructing a particular version of the SCCS file. For example

```
get –r2.3 –l s.abc
```

generates an l.file showing the deltas applied to retrieve version 2.3 of file **s.abc.** Specifying **p** with –l, as in

```
get –lp –r2.3 s.abc
```

causes the output to be written to the standard output rather than to the l.file. **get –g** can be used with **–l** to suppress the retrieval of the text.

get –m identifies the changes applied to an SCCS file. Each line of the g.file is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

get –n causes each line of a g.file to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep**(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get –p –n –s directory | grep pattern
```

If both **–m** and **–n** are specified, each line of the generated g.file is preceded by the value of the **chap3.13** ID keyword and a tab (this is the effect of **–n**) and is followed by the line in the format produced by **–m**. Because use of **–m** and/or **–n** causes the contents of the g.file to be modified, such a g.file must not be used for creating a delta. Therefore, neither **–m** nor **–n** may be specified together with **get –e**.

NOTE See **get**(1) in the *IRIS 4D Programmer's Reference Manual* for a full description of additional keyletters.

The delta Command

The **delta**(1) command is used to incorporate changes made to a g.file into the corresponding SCCS file—that is, to create a delta and, therefore, a new version of the file.

The **delta** command requires the existence of a p.file (created via **get –e**). It examines the p.file to verify the presence of an entry containing the user's login name. If none is found, an error message results.

get –e performs. If all checks are successful, **delta** determines what has been changed in the g.file by comparing it via **diff**(1) with its own temporary copy of the g.file as it was before editing. This temporary copy of the g.file is called the d.file and is obtained by performing an internal **get** on the SID specified in the p.file entry.

The required p.file entry is the one containing the login name of the user executing **delta**, because the user who retrieved the g.file must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **get –e** more than once on the same SCCS file. Then, **delta –r** must be used to specify the SID that uniquely identifies the p.file entry. This entry is then the one used to obtain the SID of the delta to be created.

In practice, the most common use of **delta** is

delta s.abc

which prompts

```
comments?
```

to which the user replies with a description of why the delta is being made, ending the reply with a newline character. The user's response may be up to 512 characters long with newlines (not intended to terminate the response) escaped by backslashes, \

If the SCCS file has a **v** flag, **delta** first prompts with

```
MRs?
```

(Modification Requests), on the standard output. The standard input is then read for MR numbers, separated by blanks and/or tabs, ended with a newline character. A

Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report, change request, trouble ticket, etc., collectively called MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change management process.

delta –**y** and/or –**m** can be used to enter comments and MR numbers on the command line rather than through the standard input, as in

delta –**y**"*descriptive comment*" –**m**"*mrnum1 mrnum2*" **s.abc**

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two keyletters are useful when **delta** is executed from within a shell procedure (see **sh**(1) in the *IRIS-4D Programmer's Reference Manual*).

NOTE

delta –**m** is allowed only if the SCCS file has a **v** flag.

No matter how comments and MR numbers are entered with **delta**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **delta**.

If **delta** is used with more than one file argument and the first file named has a v flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the p.file) and the number of lines inserted, deleted, and left unchanged. For example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the user's perception of the changes made to a g.file, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the g.file. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited g.file.

If you are in the process of making a delta, the **delta** command finds no ID keywords in the edited g.file, the message

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a g.file that was created by a **get** without **−e** (ID keywords are replaced by **get** in such a case). It could also be caused by accidentally deleting or changing ID keywords while editing the g.file. Or, it is possible that the file had no ID keywords. In any case, the delta will be created unless there is an **i** flag in the SCCS file (meaning the error should be treated as fatal), in which case the delta will not be created.

After the processing of an SCCS file is complete, the corresponding p.file entry is removed from the p.file. All updates to the p.file are made to a temporary copy, the "q.file," whose use is similar to the use of the x.file described earlier under "SCCS Command Conventions." If there is only one entry in the p.file, then the p.file itself is removed.

In addition, **delta** removes the edited g.file unless **−n** is specified. For example

```
delta -n s.abc
```

will keep the g.file after processing.

delta −s suppresses all output normally directed to the standard output, other than `comments?` and `MRs?`. Thus, use of **−s** with **−y** (and/or **−m**) causes **delta** to neither read the standard input nor write the standard output.

The differences between the g.file and the d.file constitute the delta and may be printed on the standard output by using **delta −p**. The format of this output is similar to that produced by **diff**(1).

The admin Command

The **admin**(1) command is used to administer SCCS files—that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters with **admin** or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing SCCS files.

Two keyletters are used in detecting and correcting corrupted SCCS files (see "Auditing" under "SCCS Files").

Newly created SCCS files are given access permission mode 444 (read only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

Creation of SCCS Files

An SCCS file can be created by executing the command

```
admin -ifirst s.abc
```

in which the value **first** with **–i** is the name of a file from which the text of the initial delta of the SCCS file **s.abc** is to be taken. Omission of a value with **–i** means **admin** is to read the standard input for the text of the initial delta.

The command

```
admin -i s.abc < first
```

is equivalent to the previous example.

If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the command also sets the **i** flag (not to be confused with the **–i** keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin –i**.

admin –r is used to specify a release number for the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

means the first delta should be named 3.1 rather than the normal 1.1. Because **–r** has meaning only when creating the first delta, its use is permitted only with **–i**.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may want to record why this was done. Comments (**admin –y**) and/or MR numbers (**–m**) can be entered in exactly the same way as a **delta.**

If **–y** is omitted, a comment line of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**admin –m**), the **v** flag must be set via **–f**. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary (see **sccsfile**(4) in the *IRIS-4D Programmer's Reference Manual*) in the SCCS file. Thus:

admin –ifirst –m*mrnum1* –fv s.abc

Note that –y and –**m** are effective only if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin –t**.

When an SCCS file is first being created and –**t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
admin –ifirst –tdesc s.abc
```

specifies that the descriptive text is to be taken from file **desc**.

When processing an existing SCCS file, –**t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin –tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of **desc**. Omission of the filename after the –**t** keyletter as in

```
admin –t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized or changed by **admin** –**f**, or deleted via –**d**.

SCCS file flags are used to direct certain actions of the various commands. (See **admin**(1) in the *IRIS-4D Programmer's Reference Manual* for a description of all the flags.) For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

admin –**f** is used to set flags and, if desired, their values. For example

```
admin –ifirst –fi –fmmodname s.abc
```

sets the **i** and **m** (module name) flags. The value *modname* specified for the **m** flag is the value that the **get** command will use to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the g.file is used as the replacement for the %M% ID keyword.) Several –**f** keyletters may be supplied on a single **admin**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

admin –d is used to delete a flag from an existing SCCS file. As an example, the command

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **–d** keyletters may be used with one **admin** and may be intermixed with **–f**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), **admin –a** is used. For example,

```
admin -axyz -awql -a1234 s.abc
```

adds the login names **xyz** and **wql** and the group ID **1234** to the list. **admin –a** may be used whether creating a new SCCS file or processing an existing one.

admin –e (erase) is used to remove login names or group IDs from the list.

The prs Command

The **prs**(1) command is used to print all or part of an SCCS file on the standard output. If **prs –d** is used, the output will be in a format called data specification. Data specification is a string of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example,

```
:I:
```

is defined as the data keyword replaced by the SID of a specified delta. Similarly, **:F:** is the data keyword for the SCCS filename currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list, see **prs**(1) in the *IRIS-4D Programmer's Reference Manual.*

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example,

```
prs -d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying its SID using **prs –r**. For example,

```
prs -d":F:: :I: comment line is: :C:" -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If **–r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **–l** or **–e**. The use of **prs –e** substitutes data keywords for the SID designated via **–r** and all deltas created earlier, while **prs –l** substitutes data keywords for the SID designated via **–r** and all deltas created later. Thus, the command

```
prs -d:I: -r1.4 -e s.abc
```

may output

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command

```
prs -d:I: -r1.4 -l s.abc
```

may produce

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **–e** and **–l**.

The sact Command

sact(1) is like a special form of the **prs** command that produces a report about files that are out for edit. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **get** –e command, and the date and time the **get –e** was executed. It is a useful command for an administrator.

The help Command

The **help**(1) command prints the syntax of SCCS commands and of messages that may appear on the user's terminal. Arguments to **help** are simply SCCS commands or the code numbers that appear in parentheses after SCCS messages. (If no argument is given, **help** prompts for one.) Explanatory information is printed on the standard output. If no information is found, an error message is printed. When more than one argument is used, each is processed independently, and an error resulting from one will not stop the processing of the others.

Explanatory information related to a command is a synopsis of the command. For example,

help ge5 rmdel

produces

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
  rmdel  -rSID  name  ...
```

The rmdel Command

The **rmdel**(1) command allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 11-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **–r** keyletter is mandatory with **rmdel**. It is used to specify the complete SID of the delta to be removed. Thus,

```
rmdel –r2.3 s.abc
```

specifies the removal of trunk delta 2.3.

Before removing the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation:

floor less than or equal to R less than or equal to ceiling

The **rmdel** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing. That is, if the **l** flag is set (see **admin**(1) in the *IRIS-4D Programmer's Reference Manual*), the release must not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

The cdc Command

The **cdc**(1) command is used to change the commentary made when the delta was created. It is similar to the **rmdel** command (e.g., **–r** and full SID are necessary), although the delta need not be a leaf delta. For example,

```
cdc –r3.4 s.abc
```

specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for as with **delta.**

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new and deletion of old ("!" prefix) MR numbers. Thus,

```
cdc –r1.4 s.abc
MRs? mrnum3 !mrnum1                (The MRs? prompt appears only
                                   if the v flag has been set.)
comments? deleted wrong MR number and inserted correct MR number
```

inserts **mrnum3** and deletes **mrnum1** for delta 1.4.

NOTE: An MR (Modification Request) is described above under the **delta** command.

The what Command

The **what**(1) command is used to find identifying information within any UNIX file whose name is given as an argument. No keyletters are accepted. The **what** command searches the given file(s) for all occurrences of the string @(#), which is the replacement for the *%Z%* ID keyword (see **get**(1)). It prints on the standard output whatever follows the string until the first double quote, ", greater than, >, backslash, \, newline, or nonprinting NUL character.

For example, if an SCCS file called **s.prog.c** (a C language program) contains the following line:

```
char  id[]= "%W%";
```

and the command

```
get –r3.4 s.prog.c
```

is used, the resulting g.file is compiled to produce **prog.o** and **a.out**. Then, the command

```
what prog.c prog.o a.out
```

produces

```
prog.c:
  prog.c:  3.4
prog.o:
  prog.c:  3.4
a.out:
  prog.c:  3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

The sccsdiff Command

The **sccsdiff**(1) command determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff –r** in the same way as with **get –r**. SID numbers must be specified as the first two arguments. Any following keyletters are interpreted as arguments to the **pr**(1) command (which prints the differences) and must appear before any filenames. The SCCS file(s) to be processed are named last. Directory names and a name of – (a lone minus sign) are not acceptable to **sccsdiff.**

The following is an example of the format of **sccsdiff**:

sccsdiff –r3.4 –r5.6 s.abc

The differences are printed the same way as by **diff**(1).

The comb Command

The **comb**(1) command lets the user try to reduce the size of an SCCS file. It generates a shell procedure (see **sh**(1) in the *IRIS-4D Programmer's Reference Manual*) on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 11-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the keyletters used with this command are:

comb –s	This option generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as an advance step.
comb –p	This option is used to specify the oldest delta the user wants preserved.
comb –c	This option is used to specify a list (see **get**(1) in the *IRIS-4D Programmer's Reference Manual* for its syntax) of deltas the user wants preserved. All other deltas will be discarded.

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

The val Command

The **val**(1) command is used to determine whether a file is an SCCS file meeting the characteristics specified by certain keyletters. It checks for the existence of a particular delta when the SID for that delta is specified with **–r**.

The string following –y or **–m** is used to check the value set by the **t** or **m** flag, respectively. See **admin**(1) in the *IRIS-4D Programmer's Reference Manual* for descriptions of these flags.

The **val** command treats the special argument – differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until an end-of-file (CTRL-D) is entered. This permits one **val** command with different values for keyletters and file arguments. For example,

```
val – –yc –mabc s.abc –mxyz –ypl1 s.xyz
```

first checks if file **s.abc** has a value **c** for its type flag and value **abc** for the module name flag. Once this is done, **val** processes the remaining file, in this case **s.xyz**.

The **val** command returns an 8-bit code. Each bit set shows a specific error (see **val**(1) for a description of errors and codes). In addition, an appropriate diagnostic is printed unless suppressed by –s. A return code of 0 means all files met the characteristics specified.

The vc Command

The **vc**(1) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. A complete description of **vc** may be found in the *IRIS-4D Programmer's Reference Manual.*

SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the UNIX system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files—that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings—subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating a copy of the file (the x.file; see "SCCS Command Conventions"). When processing is done, the old file is automatically removed and the x.file renamed (**s.** prefix). If the old file had additional links, this breaks them. Then, rather than process such files, SCCS commands will produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (e.g., on large development projects), one user—that is, one user ID—must be chosen as the owner of the SCCS files. This person will administer the files (e.g. use the **admin** command) and will be SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and, if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the set user ID on execution bit on (see **chmod**(1) in the *IRIS-4D User's Reference Manual*). This assures that the effective user ID is the user ID of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmdel** and **cdc**.

A project-dependent interface program, as its name implies, can be custom built for each project. Its creation is discussed later under "An SCCS Interface Program."

Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	a line containing the logical sum of all the characters of the file (not including the checksum itself)
Delta Table	information about each delta, such as type, SID, date and time of creation, and commentary
User Names	list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas
Flags	indicators that control certain actions of SCCS commands
Descriptive Text	usually a summary of the contents and purpose of the file
Body	the text administered by SCCS, intermixed with internal SCCS control lines

Details on these file sections may be found in **sccsfile**(4). The checksum is discussed below under "Auditing."

Since SCCS files are ASCII files they can be processed by non-SCCS commands like **ed**(1), **grep**(1), and **cat**(1). This is convenient when an SCCS file must be modified manually (e.g., a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when a user wants simply to look at the file.

CAUTION Extreme care should be exercised when modifying SCCS files with commands that are not SCCS.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command will issue an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed (possibly by having lost one or more blocks or by having been modified with **ed**(1)). No SCCS command will process a corrupted SCCS file except the **admin** command with **–h** or **–z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin –h** and specify all SCCS files:

> **admin –h** s.*file1* s.*file2* ...
> or
> **admin –h** *directory1 directory2* ...

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process will not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **ls**(1) command periodically, and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

> **admin –z** s.*file*

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file will no longer be detectable.

The lint Program

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions, which nevertheless are legal. **lint** accepts multiple input files and library specifications and checks them for consistency.

Using lint

The **lint** command has the form:

> **lint** [*options*] *files* ... *library-descriptors* ...

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

–c	Only check for intra-file bugs; leave external information in files suffixed with **.ln**.
–h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
–n	Do not check for compatibility with either the standard or the portable **lint** library.
–o *name*	Create a lint library from input files named **llib–l***name***.ln**.
–p	Attempt to check portability.
–u	Suppress messages about function and external variables used and not defined or defined and not used.
–x	Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as –**ab** or –**xha**.

The names of files that contain C language programs should end with the suffix .c, which is mandatory for **lint** and the C compiler.

lint accepts certain arguments, such as:

> **–lm**

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions. The next section, "**lint** Message Types," describes how it is done.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file but are not used in a source file do not result in messages. **lint** does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **–p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **–n** option can be used to suppress all library checking.

lint Message Types

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

lint prints messages about variables and functions which are defined but not otherwise mentioned, unless the message is suppressed by means of the **–u** or **–x** option.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, the **–v** option is available to suppress the printing of these messages. When **–v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

The comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

When **lint** is applied to some but not all files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is, of course, more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **–u** option suppresses the spurious messages.

Set/Used Information

lint attempts to detect cases where a variable is used before it is set. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use" since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

lint attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. **lint** also prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style.

lint has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the **–b** option when dealing with such input.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. **lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
    return( expr );
```

and

```
    return ;
```

statements is cause for alarm; **lint** will give the message

```
    function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
    f ( a ) {
              if ( a ) return ( 3 );
              g ();
              }
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced when in fact nothing is wrong. A comment

```
    /*NOTREACHED*/
```

in the source code will cause the message to be suppressed. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used.

When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void). For example:

```
        (void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

lint enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions
- in the use of enumerations

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*s can, of course, be intermixed with pointers to *x*s.

The type checking rules also require that, in structure references, the left operand of the –> be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where **p** is a character pointer. **lint** will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. Nevertheless, **lint** will not print messages about this.

Nonportable Character Use

On some systems, characters are signed quantities with a range from –128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
signed char c;
          . . .
if( (c = getchar()) < 0 ) . . .
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. Please note, **lint** does not catch these potential errors.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to suppress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message

```
null effect
```

from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. **lint** will print the message

```
degenerate unsigned comparison
```

in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
    if( x&077 == 0 ) ...
```

and

```
    x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =−, ...) could cause ambiguous expressions, such as:

```
    a =-1 ;
```

which could be taken as either

```
    a =- 1 ;
```

or

```
    a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, −=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
    int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
    int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
    int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
    int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. **lint** tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message

```
    possible pointer alignment problem
```

results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
    a[i] = b[i++];
```

will cause **lint** to print the message

```
    warning: i evaluation order undefined
```

in order to call attention to this condition.

Introduction

With the UNIX system running on smaller machines, such as the IRIS-4D computer, efficient use of disk storage space, memory, and computing power is becoming increasingly important. A shared library can offer savings in all three areas. For example, if constructed properly, a shared library can make **a.out** files (executable object files) smaller on disk storage and processes (**a.out** files that are executing) smaller in memory.

The first part of this chapter, "Using a Shared Library," is designed to help you use UNIX System V shared libraries. It describes what a shared library is and how to use one to build **a.out** files. It also offers advice about when to use and when not to use a shared library and how to determine whether an **a.out** uses a shared library.

The second part in this chapter, "Building a Shared Library," describes how to build a shared library. You do not need to read this part to use shared libraries. It addresses library developers, advanced programmers who are expected to build their own shared libraries. Specifically, this part describes how to use the UNIX system tool **mkshlib**(1) (documented in the *IRIX Programmer's Reference Manual*) and how to write C code for shared libraries on a UNIX system. An example is included. This part also describes a simple method of checking the compatibility of two versions of a shared library. Read this part of the chapter only if you have to build a shared library.

NOTE Shared libraries are a feature introduced with IRIX™ Release 3.0. An executable object file that needs shared libraries will not run on previous releases of IRIX.

Using a Shared Library

If you are accustomed to using libraries to build your applications programs, shared libraries should blend into your work easily. This part of the chapter explains shared libraries and tells how and when to use them on the IRIX system.

What Is a Shared Library?

A shared library is a file containing object code that several **a.out** files may use simultaneously while executing. When a program is compiled or link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UNIX system executes the resulting **a.out** file, it uses the information in this section to bring the required shared library code into the address space of the process.

The implementation behind these concepts is a shared library with two pieces. The first, called the host shared library, is an archive that the link editor searches to resolve user references and to create the **.lib** section in **a.out** files. The structure and operation of this archive is the same as any archive without shared library members. For simplicity, however, in this chapter references to archives mean archive libraries without shared library members.

The second part of a shared library is the target shared library. This is the file that the UNIX system uses when running **a.out** files built with the host shared library. It contains the actual code for the routines in the library. Naturally, it must be present on the the system where the **a.out** files will be run.

A shared library offers several benefits by not copying code into **a.out** files. It can

- save disk storage space

  Because shared library code is not copied into all the **a.out** files that use the code, these files are smaller and use less disk space.

- save memory

  By sharing library code at run time, the dynamic memory needs of processes are reduced.

- make executable files using library code easier to maintain

  As mentioned above, shared library code is brought into a process' address space at run time. Updating a shared library effectively updates all executable files that use the library, because the operating system brings the updated version into new processes. If an error in shared library code is fixed, all processes automatically use the corrected code.

Archive libraries cannot, of course, offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

"Deciding Whether to Use a Shared Library" in this chapter describes shared libraries in more detail.

The UNIX System Shared Libraries

Shared versions of the IRIX C library and IRIX graphics library are provided with IRIX Release 3.1 and later.

Shared Library	Host Library Command Line Option	Target Library Pathname
C Library	**–lc_s**	**/lib/libc_s**
Graphics Library	**–lgl_s**	**/usr/lib/libgl_s**
Font Manager Library	**–lfm_s**	**/usr/lib/libfm_s**

Notice the _s suffix on the library names; we use it to identify both host and target shared libraries. For example, it distinguishes the standard relocatable C library **libc** from the shared C library **libc_s**. The _s also indicates that the libraries are statically linked.

The relocatable C library is still available on IRIX releases; this library is searched by default during the compilation or link editing of C programs. All other archive libraries from previous releases of the system are also available. Just as you use the archive libraries' names, you must use a shared library's name when you want to use it to build your **a.out** files. You tell the link editor its name with the **–l** option, as shown below.

Building an a.out File

You direct the link editor to search a shared library the same way you direct a search of an archive library on the UNIX system:

cc *file*.**c** **–o** *file* ... **–l***library_file* ...

To direct a search of the graphics library, for example, use the following command line.

```
cc file.c -ofile ... -lgl_s ...
```

And to link all the files in your current directory together with the shared C library you'd use the following command line:

```
cc *.c -lc_s
```

Normally, you should include the **-lc_s** argument after all other **-l** arguments on a command line. The shared C library will then be treated like the relocatable C library, which is searched by default after all other libraries specified on a command line are searched.

Coding an Application

Application source code in C or assembly language is compatible with both archive libraries and shared libraries. As a result, you should not have to change the code in any applications you already have when you use a shared library with them. When coding a new application for use with a shared library, you should just observe your standard coding conventions.

However, do keep the following two points in mind, which apply when using either an archive or a shared library:

- Don't define symbols in your application with the same names as those in a library.

  Although there are exceptions, you should avoid redefining standard library routines, such as **printf**(3S) and **strcmp**(3C). Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.

- Don't use undocumented archive routines.

  Use only the functions and data mentioned on the manual pages describing the routines in Section 3 of the *IRIX Programmer's Reference Manual.* For example, don't try to outsmart the **ctype** design by manipulating the underlying implementation.

Deciding Whether to Use a Shared Library

You should base your decision to use a shared library on whether it saves space in disk storage and memory for your program. A well-designed shared library almost always saves space. So, as a general rule, use a shared library when it is available.

To determine what savings are gained from using a shared library, you might build the same application with both an archive and a shared library, assuming both kinds are available. Remember, that you may do this because source code is compatible between shared libraries and archive libraries. (See the above section "Coding an Application.") Then compare the two versions of the application for size and performance. For example,

```
$ cat hello.c
main()
{
   printf("Hello\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc_s
$ size unshared shared
text      data      bss       dec       hex
1000      1000      3da0      23968     5da0      shared
4000      1000      44f0      38128     94f0      unshared
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. The following section gives a more detailed discussion about when a shared library does and does not save space.

More About Saving Space

This section is designed to help you better understand why your programs will usually benefit from using a shared library. It explains

- how shared libraries save space that archive libraries cannot
- how shared libraries are implemented on the UNIX system
- how shared libraries might increase space usage

How Shared Libraries Save Space

To better understand how a shared library saves space, we need to compare it to an archive library.

A host shared library resembles an archive library in three ways. First, as noted earlier, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside and made visible outside the library are external symbols. Note that the library may also have imported symbols, symbols that it uses but usually does not define. Third, the link editor searches the library for these symbols when linking a program to resolve its external references. By resolving the references, the link editor produces an executable version of the program, the **a.out** file.

NOTE Note that the link editor on the UNIX system is a static linking tool; static linking requires that all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both an archive library and a shared library.

Although these similarities exist, a shared library differs significantly from an archive library. The major differences relate to how the libraries are handled to resolve symbolic references, a topic already discussed briefly.

Consider how the UNIX system handles both types of libraries during link editing. To produce an **a.out** file using an archive library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor copies from the shared library into the program's object file only a small amount of code for initialization of imported symbols. (See the section "Importing Symbols" later in the chapter for more details on imported symbols.) For the bulk of the library code, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values.

When the UNIX system executes the resulting **a.out** file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

Figure 13-1 depicts the **a.out** files produced using a regular archive version and a shared version of the standard C library to compile the following program:

```
main()
{
    ...
    printf( "How do you like this manual?\n" );
    ...
    result = strcmp( "I do.", answer );
    ...
}
```

Notice that the shared version is smaller. Figure 13-2 depicts the process images in memory of these two files when they are executed.

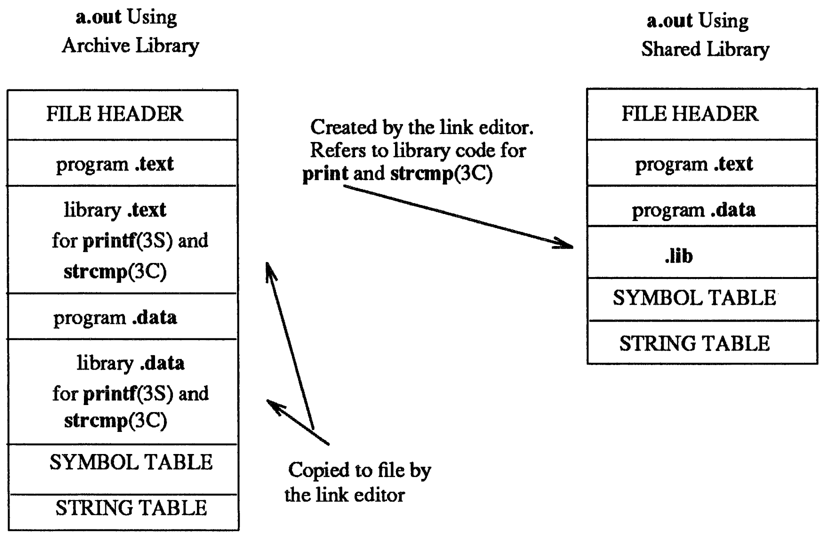

a.out Using
Archive Library
FILE HEADER
program **.text**
library **.text** for **printf**(3S) and **strcmp**(3C)
program **.data**
library **.data** for **printf**(3S) and **strcmp**(3C)
SYMBOL TABLE
STRING TABLE
Created by the link editor. Refers to library code for **print** and **strcmp**(3C)
Copied to file by the link editor
a.out Using
Shared Library
FILE HEADER
program **.text**
program **.data**
.lib
SYMBOL TABLE
STRING TABLE

Figure 13-1: **a.out** Files Created Using an Archive Library and a Shared Library

Now consider what happens when several **a.out** files need the same code from a library. When using an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the **a.out** files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the **a.out** files, as indicated in Figure 13-2. This separation enables all processes using the same shared library to reference a single copy of the code.

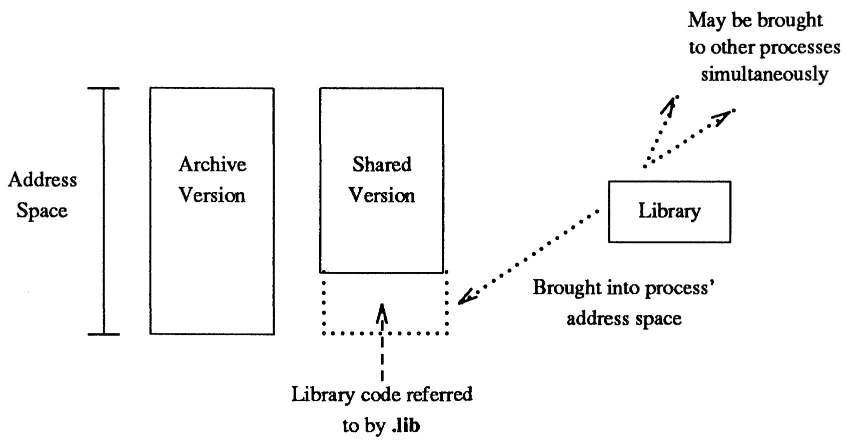

May be brought to other processes simultaneously
Address Space
Archive Version
Shared Version
Library
Brought into process' address space
Library code referred to by .lib

Figure 13-2: Processes Using an Archive and a Shared Library

How Shared Libraries Are Implemented

Now that you have a better understanding of how shared libraries save space, you need to consider their implementation on the UNIX system to understand how they might increase space usage (this happens seldomly). The following paragraphs describe host and target shared libraries, the branch table, and then how shared libraries might increase space usage.

The Host Library and Target Library

As previously mentioned, every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which you build an **a.out** file; the target machine is the machine on which you run the file. Of course, the host and target may be the same machine, but they don't have to be.

The host library is just like an archive library. Each of its members (typically a complete object file) defines some text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation or link editing of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, as mentioned earlier, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the **a.out** file mentioned earlier (see the section "What is a Shared Library?") and shown in Figure 13-1 as **.lib.**

The target library used for execution resembles an **a.out** file. The UNIX operating system reads this file during execution if a process needs a shared library. The special **.lib** section in the **a.out** file tells which shared libraries are needed. When the UNIX system executes the **a.out** file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, which is where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them. That is, the UNIX system uses the same physical code for each process that attaches a shared library's text.

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region (contiguous area of virtual address space that mirrors the **.data** section of the target library). Processes that share text do not share data and stack area so that they do not interfere with one another.

As suggested above, the target library is a lot like an **a.out** file, which can also share its text, but not its data. Also, a process must have execute permission for a target library to execute an **a.out** file that uses the library.

The Branch Table

When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. This is because a static linking loader like **ld** binds symbols to addresses during link editing. In this way, the **a.out** file for the program has an address for each referenced symbol.

What happens if library code is updated and the address of a symbol changes? Nothing happens to an **a.out** file built with an archive library, because that file already has a copy of the code defining the symbol. (Even though it isn't the updated copy, the **a.out** file will still run.) However, the change can adversely affect an **a.out** file built with a shared library. This file has only a symbol telling where the required library code is. If the library code were updated, the location of that code might change. Therefore, if the **a.out** file ran after the change took place, the operating system could bring in the wrong code. To keep the **a.out** file current, you might have to recompile a program that uses a shared library after each library update.

To prevent the need to recompile, a shared library is implemented with a branch table on the UNIX system. A branch table associates text symbols with absolute addresses that do not change even when library code is changed. Each address labels a jump instruction to the address of the code that defines a symbol.

Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 13-3 shows two **a.out** files executing that make a call to **printf**(3S). The process on the left was built using an archive library. It already has a copy of the library code defining the **printf**(3S) symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

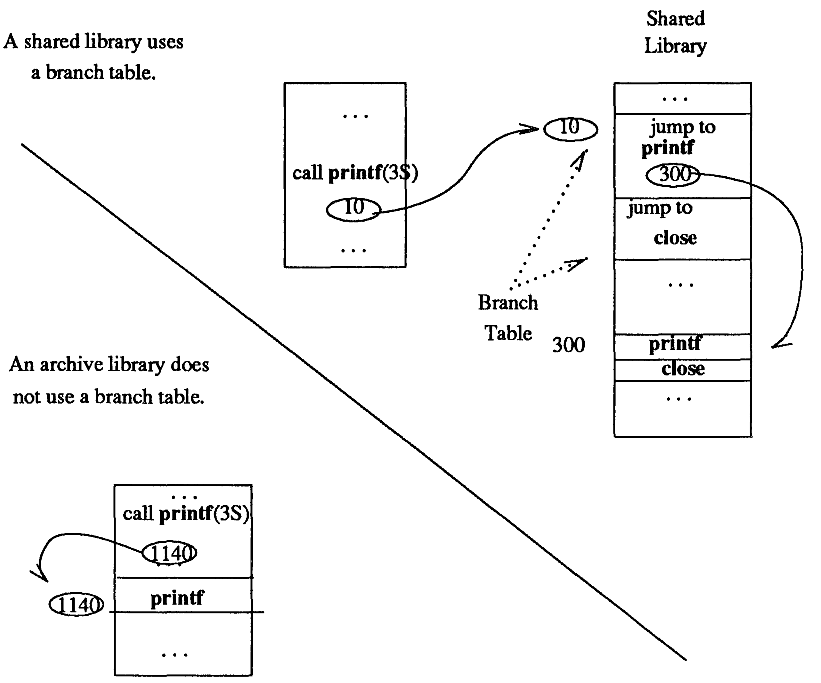


Figure 13-3: A Branch Table in a Shared Library

Data symbols do not have a mechanism to prevent a change of address between shared libraries. Therefore, you must take great care when designing your shared library to ensure that the addresses of global data symbols can be kept constant in future versions of the shared library.

How Shared Libraries Might Increase Space Usage

A target library might add space to a process. Recall from "How Shared Libraries are Implemented" in this chapter that a shared library's target file may have both text and data regions connected to a process. While the text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. Naturally, this region adds to the process' memory requirements. As a result, if an application uses only a small part of a shared library's text and data, executing the application might require more memory with a shared library than without one. For example, it would be unwise to use the shared C library to access only **strcmp**(3C). Although sharing **strcmp**(3C) saves disk storage and memory, the memory cost for sharing all the shared C library's private data region outweighs the savings. The archive version of the library would be more appropriate.

A host library might add space to an **a.out** file. Recall that UNIX System V Release 3.0 uses static linking, which requires that all external references in a program be resolved before it is executed. Also recall that a shared library may have imported symbols, which are used but not defined by the library. To resolve these references, the link editor has to add to the **a.out** initialization code defining the referenced imported symbols file. This code increases the size of the **a.out** file.

Identifying a.out Files that Use Shared Libraries

Suppose you have an executable file and you want to know whether it uses a shared library. You can use the **odump**(1) command (documented in the *IRIX Programmer's Reference Manual*) to look at the section headers for the file:

```
odump –hv a.out
```

If the file has a **.lib** section, a shared library is needed. If the **a.out** does not have a **.lib** section, it does not use shared libraries.

With a little more work, you can even tell what libraries a file uses by looking at the **.lib** section contents.

```
odump –L a.out
```

Building a Shared Library

This part of the chapter explains how to build a shared library. It covers the major steps in the building process, the use of the UNIX system tool **mkshlib**(1) that builds the host and target libraries, and some guidelines for writing shared library code. There is an example at the end of this part which demonstrates the major features of **mkshlib** and the steps in the building process.

This part assumes that you are an advanced C programmer faced with the task of building a shared library. It also assumes you are familiar with the archive library building process. You do not need to read this part of the chapter if you only plan to use the UNIX system shared libraries or other shared libraries that have already been built.

The Building Process

To build a shared library on the UNIX system, you have to complete six major tasks:

- Choose region addresses.
- Choose the pathname for the shared library target file.
- Select the library contents.
- Rewrite existing library code to be included in the shared library.
- Write the library specification file.
- Use the **mkshlib** tool to build the host and target libraries.

Each of these tasks is discussed below.

Step 1: Choosing Region Addresses

The first step in building a shared library is to choose its region addresses.

Your shared library will need two regions — one for its **text** and one for its **data.** Regions on IRIS-4D are necessarily the same size as the size of first-level TLB entries. Currently, this size is 2Mb, but it is possible that it will be increased to 4Mb at some future date. Thus, a region begins on each 4Mb boundary. In addition, the architecture of the MIPS processor forces the text region to be in the lower 256Mb, **and** the data segment of the shared library to be at a lower address than other data. This forces the sharing of the lowest 256Mb area between all program and library text, all system shared libraries, and all user shared libraries.

To avoid conflicts, use the following rules when choosing the region addresses of your shared library:

- Each region address should be on a 4Mb boundary.
- The two regions should be *adjacent*, with the address of the *text* region lower than the address of the *data* region.
- The regions should be at the highest address possible **below** 0x0c000000.

NOTE Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single **a.out** file doesn't need to use both libraries.

If you are building a large system with many **a.out** files and processes, shared libraries might improve its performance. As long as you don't intend to release the shared libraries as separate products, you may choose any region addresses below 0x0c000000 that are not in use on your system. If you plan to release the library, you must consult with the concerned parties to agree on region addresses. Don't risk address conflicts.

NOTE If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well. Some of your customers might not find the shared library appropriate for their applications. Others might want their applications to run on versions of the UNIX system without shared library support.

Step 2: Choosing the Target Library Pathname

After you choose the region addresses for your shared library, you should choose the pathname for the target library. We chose **/lib/libc_s** for the shared C library, **/usr/lib/libgl_s** for the graphics library, and **/usr/lib/libfm_s** for the font manager library. (As mentioned earlier, we use the _s suffix in the pathnames of all statically linked shared libraries.) To choose a pathname for your shared library, consult the established list of names for your computer or see your system administrator. Also keep in mind that shared libraries needed to boot a UNIX system should normally be located in /**lib**; other application libraries normally reside in /**usr/lib** or in private application directories. Of course, if your shared library is for personal use, you can choose any convenient pathname for the target library.

Step 3: Selecting Library Contents

Selecting the contents for your shared library is the most important task in the building process. Some routines are prime candidates for sharing; others are not. For example, it's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. What you include will depend on the individual needs of the programmers and other users for whom you are building the library. There are some general guidelines you should follow, however. They are discussed in the section "Choosing Library Members" in this chapter. Also see the guidelines in the following sections: "Importing Symbols" and "Tuning the Shared Library Code."

Step 4: Rewriting Existing Library Code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See the section "Changing Existing Code for the Shared Library" in this chapter.

Step 5: Writing the Library Specification File

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. An example specification file is given in the section towards the end of the chapter, "An Example." Also, see the section "Using the Specification File for Compatibility" in this chapter. The contents and format of the specification file are given by the following directives (see also the **mkshlib**(1) manual page).

All directives that are followed by multi-line specifications are valid until the next directive or the end of file.

#address *sectname address*

Specifies the start address, *address*, of section *sectname* for the target file. This directive is typically used to specify the start addresses of the **.text** and **.data** sections.

#target *pathname*

Specifies the pathname, *pathname*, of the target shared library on the target machine. This is the location where the operating system looks for the shared library during execution. Normally, *pathname* will be an absolute pathname, but it does not have to be.

This directive must be specified exactly once per specification file.

#branch	Starts the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

funcname is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. *position* may be a single integer or a range of integers of the form *position1-position2*. Each *position* must be greater than or equal to one. The same position cannot be specified more than once, and every position from one to the highest given position must be accounted for.

If a symbol is given more than one branch table entry by associating a range of positions with the symbol or by specifying the same symbol on more than one branch table specification line, then the symbol is defined to have the address of the highest associated branch table entry. All other branch table entries for the symbol can be thought of as empty slots and can be replaced by new entries in future versions of the shared library.

Finally, only functions should be given branch table entries, and those functions must be external.

This directive must be specified exactly once per shared library specification file.

#objects Specifies the names of the object files constituting the target shared library. The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each filename followed by a newline character. This list of objects will be used to build the shared library.

These modules must be compiled with the -G 0 option; do not compile using gp-region.

This directive must be specified exactly once per shared library specification file.

#init *object* Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

ptr <white space> *import*

ptr is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each such line is of the form:

ptr = &import;

All initializations for a particular object file must be given once and multiple specifications of the same object file are not allowed.

#ident *string* — Specifies a string, *string*, to be included in the **.comment** section of the target shared library and the **.comment** sections of every member of the host shared library.

— Specifies a comment. The rest of the line is ignored.

Step 6: Using mkshlib to Build the Host and Target

The UNIX system command **mkshlib**(1) builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as**(1), and link editor, **ld**(1). Tools are invoked through the use of **execvp** (see **exec**(2)), which searches directories in a user's $PATH environment variable. These commands all are documented in the *IRIX Programmer's Reference Manual.*

The user input to **mkshlib** consists of the library specification file and command line options. We just discussed the specification file; let's take a look at the options now. The shared library build tool has the following syntax:

mkshlib –**s** *specfil* –**t** *target* [–**h** *host*] [–**n**] [–**q**]

–s *specfil* — Specifies the shared library specification file, *specfil*. This file contains all the information necessary to build a shared library.

–**t** *target* — Specifies the name, *target*, of the target shared library produced on the host machine. When *target* is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section "Writing the Library Specification File"). If the –**n** option is given, then a new target shared library will not be generated.

–h *host*	Specifies the name of the host shared library, *host*. If this option is not given, then the host shared library will not be produced.
–n	Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **–t** option must still be supplied since a version of the target shared library is needed to build the host shared library.
–q	Suppresses the printing of certain warning messages.

Guidelines for Writing Shared Library Code

Because the main advantage of a shared library over an archive library is sharing and the space it saves, these guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility.

It's best to read these guidelines once from beginning to end to get a perspective of the things you need to consider when building a shared library. Then use the guidelines as a checklist during planning and decision-making.

There are restrictions to building a shared library, which involve static linking. Here's a summary of the restrictions; some of them are discussed in more detail later. Keep these restrictions in mind when reading the guidelines in this section.

- External symbols have fixed addresses.

  If an external symbol moves, you have to relink all **a.out** files that use the library. This restriction applies both to text and data symbols.

- If the library's text changes for one process at run time, it changes for all processes.

- If the library uses a symbol directly, that symbol's run time value (address) must be known when the library is built.

- Imported symbols cannot be referenced directly.

  Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

- Modules comprising the library must be compiled as **–G 0**.

Choosing Library Members

Include Large, Frequently Used Routines

These routines are prime candidates for sharing. Placing them in a shared library saves code space for individual **a.out** files and saves memory, too, when several concurrent processes need the same code. **printf**(3S) and related C library routines (which are documented in the *IRIX Programmer's Reference Manual*) are good examples.

The **printf**(3S) family of routines is used frequently. Therefore, **printf**(3S) and related routines have been included in the shared C library.

Exclude Infrequently Used Routines

Putting these routines in a shared library can degrade performance, particularly on paging systems. Traditional **a.out** files contain all code they need at run time. By definition, the code in an **a.out** file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single **a.out** file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance. See also "Organize to Improve Locality."

Exclude Routines that Use Much Static Data

These modules increase the size of processes. As "How Shared Libraries Are Implemented" and "Deciding Whether to Use a Shared Library" explain, every process that uses a shared library gets its own private copy of the library's data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set.

For example, **getgrent**(3C), which is documented in the *IRIX Programmer's Reference Manual*, is not used by many standard UNIX commands. Some versions of the module define over 1400 bytes of unshared, static data. It probably should not be included in a shared library. You can import global data, if necessary, but not local, static data.

Exclude Routines that Complicate Maintenance

All external symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols don't have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of external data may affect symbol addresses and break compatibility.

Include Routines the Library Itself Needs

It usually pays to make the library self-contained. For example, **printf**(3S) requires much of the standard I/O library. A shared library containing **printf**(3S) should contain the rest of the standard I/O routines, too.

NOTE

This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Changing Existing Code for the Shared Library

All C code that works in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for archive libraries (although it may be slightly bigger and slower). The guidelines explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data.

Minimize Global Data

All external data symbols are, of course, visible to applications. This can make maintenance difficult. You should try to reduce global data, as described below.

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant.

Third, allocate buffers at run time instead of defining them at compile time. This does two important things. It reduces the size of the library's data region for all processes and, therefore, saves memory; only the processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Define Text and Global Data in Separate Source Files

Separating text from global data makes it easier to prevent data symbols from moving. If new external variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Archive libraries let the link editor extract individual members. This sometimes encourages programmers to define related variables and text in the same source file. This works fine for relocatable files, but shared libraries have a different set of restrictions. Suppose external variables were scattered throughout the library modules. Then external and static data would be intermixed. Changing static data, such as a string, like **hello** in the following example, moves subsequent data symbols, even the external symbols:

```
Before                     Broken Successor

int head = 0;              int head = 0;
func()                     func()
{                          {
   ...                        ...
   p = "hello";               p = "hello, world";
   ...                        ...
}                          }
int tail = 0;              int tail = 0;
```

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus might be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

NOTE

The compilation system sometimes defines and uses static data invisibly to the user (e.g., tables for switch statements). Therefore, it is a mistake to assume that because you declare no static data in your shared library that you can ignore the guideline in this section.

Adding new external variables to a shared library may change the addresses of static symbols, but this doesn't affect compatibility. An **a.out** file has no way to reference static library symbols directly, so it cannot depend on their values. Thus it pays to group all external data symbols and place them at lower addresses than the static (hidden) data. You can write the specification file to control this. In the list of object files, make the global data files first.

```
#objects
   data1.o
   ...
   lastdata.o
   text1.o
   text2.o
   ...
```

If the data modules are not first, a seemingly harmless change (such as a new string literal) can break existing **a.out** files.

Shared library users get all library data at run time, regardless of the source file organization. Consequently, you can put all external variables' definitions in a single source file without a space penalty.

Initialize Global Data

Initialize external variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. **mkshlib** will give a fatal error if it finds an uninitialized external symbol.

Preserve Branch Table Order

You should add new functions only at the end of the branch table. After you have a specification file for the library, try to maintain compatibility with previous versions. You may add new functions without breaking old **a.out** files as long as previous assignments are not changed. This lets you distribute a new library without having to relink all of the **a.out** files that used a previous version of the library.

Importing Symbols

Normally, shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 13-4, the symbols **_libc.ptr1** and **_libc.ptr2** are imported from user's code and the symbol **_libc_malloc** from the library itself.

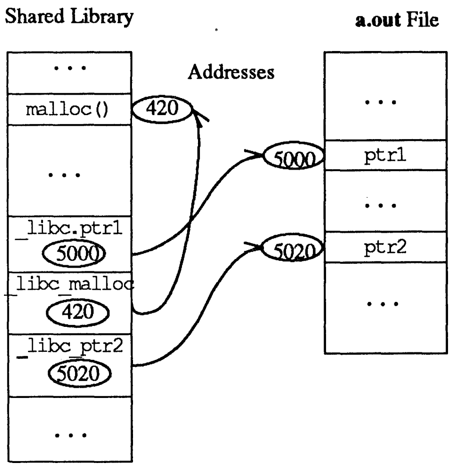


Figure 13-4: Imported Symbols in a Shared Library

The following guidelines describe when and how to use imported symbols.

Imported Symbols That the Library Does Not Define

Archive libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an **a.out** file. Neither target shared libraries nor **a.out** files can have unresolved references to symbols.

Consequently, shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing archive libraries. For the reasons stated above, it may not be appropriate to include all the archive's modules in the target shared library. Remember though that if you exclude a symbol from the target shared library that is referenced from the target shared library, you will have to import the excluded symbol.

Imported Symbols That Users Must Be Able to Redefine

Optionally, shared libraries can import their own symbols. At first this might appear to be an unnecessary complication, but consider the following. Two standard libraries, **libc** and **libmalloc**, provide a **malloc** family. Even though most UNIX commands use the **malloc** from the C library, they can choose either library or define their own.

Three possible strategies existed for the building of the shared C library. First, the **malloc**(3C) family could have been excluded. Other library members would have needed it, and so it would have been an imported symbol. This would have worked, but it would have meant less savings.

Second, the **malloc** family could have been included but not imported. This would have provided more savings for typical commands, but it had a price. Other library routines call **malloc** directly, and those calls could not have been overridden. If an application tried to redefine **malloc**, the library calls would not have used the alternate version. Furthermore, the link editor would have found multiple definitions of **malloc** while building the application. To resolve this the library developer would have to change source code to remove the custom **malloc**, or the developer would have to refrain from using the shared library.

Finally, **malloc** could have been included in the shared library, but treated as an imported symbol. This is what was done. Even though **malloc** is in the library, nothing else there refers to it directly; all references are through an imported symbol pointer. If the application does not redefine **malloc**, both application and library calls are routed to the library version. All calls are mapped to the alternate, if present.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and time overhead to the library, the technique allows a shared library to be one hundred percent compatible with an existing archive at link time and run time.

Mechanics of Importing Symbols

Assume that a shared library wants to import the symbol **malloc**. The original archive code and the shared library code appear below.

```
Archive Code                   Shared Library Code

                           /* See pointers.c on next page */

extern char *malloc();         extern char *(*_libc_malloc)();

export()                       export()
{                              {
   ...                            ...
   p = malloc(n);                 p = (*_libc_malloc)(n);
   ...                            ...
}                              }
```

Making this transformation is straightforward, but two sets of source code would be necessary to support both an archive and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The **–I** flag to **cc**(1) would direct the C preprocessor to look in the appropriate directory to find the desired file.

Archive **import.h**

```
/* empty */
```

Shared **import.h**

```
/*
 *  Macros for importing
 *  symbols.  One #define
 *  per symbol.
 */


 ...
 #define malloc    (*_libc_malloc)
 ...
 extern char *malloc();
 ...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirections for imported symbols. The declaration of **malloc** in **import.h** actually declares the pointer **_libc_malloc.**

Common Source

```
#include "import.h"

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Alternatively, one can hide the #include with #ifdef:

Common Source

```
#ifdef SHLIB
#     include "import.h"
#endif

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Of course the transformation is not complete. You must define the pointer **_libc_malloc.**

File **pointers.c**

```
char *(*_libc_malloc)() = 0;
```

Note that **_libc_malloc** is initialized to zero, because it is an external data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of **malloc** must be assigned to **_libc_malloc.** Tools that build the shared library generate the initialization code according to the library specification file.

Pointer Initialization Fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the **a.out** file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches **main.** If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

NOTE

Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into **a.out** files to set imported pointers to their correct values.

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set **_libc_malloc** to the address of **malloc**:

```
#init pmalloc.o
_libc_malloc    malloc
```

When **mkshlib** builds the host library, it modifies the file **pmalloc.o**, adding relocatable code to perform the following assignment statement:

```
_libc_malloc = &malloc;
```

When the link editor extracts **pmalloc.o** from the host library, the relocatable code goes into the **a.out** file. As the link editor builds the final **a.out** file, it resolves the unresolved references and collects all initialization fragments. When the **a.out** file is executed, the run time startup routines execute the initialization fragments to set the library pointers.

Selectively Loading Imported Symbols

Defining fewer pointers in each archive member increases the granularity of symbol selection and can prevent unnecessary objects and initialization code from being linked into the **a.out** file. For example, if an archive member defines three pointers to imported symbols, the link editor will require definitions for all three symbols, even though only one might be needed.

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the reference to the symbol.

Consider the following example. In the coarse method, a single source file might define all pointers to imported symbols:

Old **pointers.c**

```
int (*_libc_ptr1)() = 0;
char *(*_libc_malloc)() = 0;
int (*_libc_ptr2)() = 0;
...
```

Allowing the loader to resolve only those references that are needed requires multiple source files and archive members. Each of the new files defines a single pointer:

File	Contents
ptr1.c	`int (*_libc_ptr1)() = 0;`
pmalloc.c	`char *(*_libc_malloc)() = 0;`
ptr2.c	`int (*_libc_ptr2)() = 0;`

Using the three files ensures that the link editor will only look for definitions for imported symbols and load in the corresponding initialization code in cases where the symbols are actually used.

Providing Archive Library Compatibility

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes. Perhaps the best way to explain this guideline is by example:

The developers of the shared C library had an existing archive library to use as the base. This supplied code for individual routines, and a model to use for the shared library itself.

The developers wanted the host library archive file to be compatible with the relocatable archive C library. However, they did not want the shared library target file to include all routines from the archive: including them all would have hurt performance.

These goals were reached by building the host library in two steps. First, they used the available shared library tools to create the host library to match exactly the target. The resulting archive file was not compatible with the archive C library at this point. Second, they added to the host library the set of relocatable objects residing in the archive C library that were missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared C libraries compatible.

Tuning the Shared Library Code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems, such as UNIX System V Release 3.0. The suggestions come from the experience of building the shared C library.

The archive C library contains several diverse groups of functions. Many processes use different combinations of these groups, making the paging behavior of any shared C library difficult to predict. A shared library should offer greater benefits for more homogeneous collections of code. For example, a data base library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies were more predictable.

Profile the Code

To begin, profile the code that might go into the shared library.

Choose Library Contents

Based on profiling information, make some decisions about what to include in the shared library. **a.out** file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See "Choosing Library Members" in this chapter for more information.

Organize to Improve Locality

When a function is in **a.out** files, it probably resides in a page with other code that is used more often (see "Exclude Infrequently Used Routines"). Try to improve locality of reference by grouping dynamically related functions. If every call of **funcA** generates calls to **funcB** and **funcC**, try to put them in the same page. **cflow**(1) (documented in the *IRIX Programmer's Reference Manual*) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for Paging

The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Once again, an example might best explain this guideline:

The architecture of the IRIS-4D computer currently uses 4Kb pages. Using name lists and disassemblies of the shared library target file, the library developers determined where the page boundaries fell.

After grouping related functions, they broke them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Then the developers used the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreased.

After determining the branch table, they rearranged the library's object files without breaking compatibility. The developers put frequently used, unrelated functions together, because they would be called randomly enough to keep the pages in memory. System calls went into another page as a group, and so on.

The following example shows how to change the order of the library's object files:

```
Before              After

  #objects            #objects
      ...                 ...
      printf.o            strcmp.o
      fopen.o             malloc.o
      malloc.o            printf.o
      strcmp.o            fopen.o
      ...                 ...
```

Checking for Compatibility

The following guidelines explain how to check for upward-compatible shared libraries. Note, however, that upward compatibility may not always be an issue. Consider a shared library that is one piece of a larger system and is not delivered as a separate product. In this restricted case, you can identify all **a.out** files that use a particular library. As long as you rebuild all the **a.out** files every time the library changes, the **a.out** files will run successfully, even though versions of the library are not compatible. This may complicate development, but it is possible.

Checking Versions of Shared Libraries

As discussed previously, maintaining the order of the branch table and using space reserved at the end of the table to add new entries ensures compatibility of new versions of your shared library with existing code that uses it. This is not true with global data symbols. Location constancy of these symbols must be checked by hand between two versions of a shared library. This involves the generation of a symbol list for each version of the shared library using *nm*(1), and then comparing the results using *diff*(1).

The following example generates the appropriate symbol list for two versions of the shared C library and compares them:

```
nm -Bgn libc_s.new | egrep -v ' T ' > /tmp/libc_s.new
nm -Bgn libc_s.old | egrep -v ' T ' > /tmp/libc_s.old
diff -b /tmp/libc_s.old /tmp/libc_s.new
```

Incompatibilities in the location of global data symbols may be caused by an increase in the size of one of them. To avoid this, pad any non-scalar global data for future expansion.

Dealing with Incompatible Libraries

When you determine that a newer version of a library can't replace the older version, you have to deal with the incompatibility. You can deal with it in one of two ways. First, you can rebuild all the **a.out** files that use your library. If feasible, this is probably the best choice. Unfortunately, you might not be able to find those **a.out** files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target pathname to the new version of the library. The host and target pathnames are independent; so you don't have to change the host library pathname. New **a.out** files will use your new target library, but old **a.out** files will continue to access the old library.

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library pathname for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, **a.out** files that use your library will not work properly.

NOTE You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

An Example

This section contains an example of a small specialized shared library and the process by which it is created from original source and built.

The Original Source

The name of the library to be built is **libmaux** (for math auxiliary library). The interface consists of three functions:

logd	floating-point logarithm to a given base; defined in the file **log.c**
polyd	evaluate a polynomial; defined in the file **poly.c**
maux_stat	return usage counts for the other two routines in a structure; defined in **stats.c**,

an external variable:

mauxerr	set to non-zero if there is an error in the processing of any of the functions in the library and set to zero if there is no error (unlike **errno** in the C library),

and a header file:

maux.h	declares the return types of the function and the structure returned by **maux_stat**.

The source files before any modifications for inclusion in a shared library are given below.

```
/* log.c */
#include "maux.h"
#include <math.h>

/*
 * Return the log of "x" relative to the base "a".
 *
 *       logd(base, x) := log(x) / log(base);
 * where "log" is "log to the base E".
 */

double logd(base, x)
double base, x;
{
         extern int stats_logd;
         extern int total_calls;

         double logbase;
         double logx;

         total_calls++;
         stats_logd++;

         logbase = log((double)base);
         logx = log((double)x);
         if(logbase == -HUGE || logx == -HUGE) {
                  mauxerr = 1;
                  return(0);
         }
         else
                  mauxerr = 0;
         return(logx/logbase);
}
```

Figure 13-5: File log.c

```
/* poly.c */

#include "maux.h"
#include <math.h>

/* Evaluate the polynomial
 *       f(x) := a[0] * (x ^ n) + a[1] * (x ^ (n-1)) + ... + a[n];
 * Note that there are N+1 coefficients!
 * This uses Horner's Method, which is:
 *       f(x) := ((((a[0]*x) + a[1])*x) + a[2]) + ...) + a[n];
 * It's equivalent, but uses many less operations and is more precise.  */

double polyd(a, n, x)
double a[];
int n;
double x;
{
        extern int stats_polyd;
        extern int total_calls;
        double result;
        int i;

        total_calls++;
        stats_polyd++;
        if (n < 0) {
                mauxerr = 1;
                return(0);
        }
        result = a[0];
        for (i = 1; i <= n; i++)
        {
                result *= (double)x;
                result += (double)a[i];
        }
        mauxerr = 0;
        return(result);
}
```

Figure 13-6: File **poly.c**

```
/* stats.c */
#include "maux.h"

int total_calls;
int stats_logd;
int stats_polyd;

int mauxerr;

/* Return structure with usage stats for functions in library
 *    or 0 if space cannot be allocated for the structure */
struct mstats *
maux_stat()
{
         extern char * malloc();
         struct mstats * st;

         if((st = (struct mstats *) malloc(sizeof(struct mstats))) == 0)
                  return(0);
         st->st_polyd = stats_polyd;
         st->st_logd = stats_logd;
         st->st_total = total_calls;
         return(st);
}
```

Figure 13-7: File **stats.c**

```
/* maux.h */

struct mstats {
        int st_polyd;
        int st_logd;
        int st_total;
};

extern double polyd();
extern double logd();
extern struct mstats *  maux_stat();

extern int mauxerr;
```

Figure 13-8: Header File **maux.h**

Choosing Region Addresses and the Target Pathname

To begin, choose the region addresses for the library's **.text** and **.data** sections from the segments reserved for private use on the IRIS-4D computer; note that the region addresses must be on a segment boundary (4Mb):

```
.text     0x0b800000
.data     0x0bc00000
```

Also, choose the pathname for the target library:

```
/my/directory/libmaux_s
```

Selecting Library Contents

This example is for illustration purposes and so will include everything in the shared library. In a real case, it is unlikely that you would make a shared library with these three small routines unless you had many programmers using them frequently.

Rewriting Existing Code

According to the guidelines given earlier in the chapter, you need to define text and global data in separate source files. You cannot make **total_calls, stats_logd**, and **stats_polyd** *static*, as they must be visible to multiple files within the library. Hence, you must remove them from *stats.c* and place them in a separate file. In addition, the variable **mauxerr** must be removed from the same file. You also must initialize these variables to zero since shared libraries cannot have any uninitialized variables.

Next, notice that there are some references to symbols that you do not define in our shared library (i.e. **log** and **malloc**). You can import these symbols. To do so, create a new header file, **import.h**, which will be included in each of **log.c**, **poly.c**, and **stats.c**. The header file defines C preprocessor macros for these symbols to make transparent the use of indirection in the actual C source files. Use the **_libmaux_** prefixes on the pointers to the symbols because those pointers are made external, and the use of the library name as a prefix helps prevent name conflicts.

```
/* New header file import.h */
#define malloc     (*_libmaux_malloc)
#define log        (*_libmaux_log)

extern char * malloc();
extern double log();
```

Now, define the imported symbol pointers somewhere. You have already created a file for global data **maux_defs.c**, so we will add the definitions to it.

```
/* Data file maux_defs.c */

int mauxerr = 0;
double (*_libmaux_log)() = 0;
char * (*_libmaux_malloc)() = 0;
int total_calls = 0;
int stats_logd = 0;
int stats_polyd = 0;
```

Writing the Specification File

This is the specification file for **libmaux**:

```
 1  ##
 2  ## libmaux.sl - libmaux specification lfile
 3  #address .text 0x0b800000
 4  #address .data 0x0bc00000
 5  #target /my/directory/libmaux_s
 6  #branch
 7       polyd           1
 8       logd            2
 9       maux_stat       3
10  #objects
11       maux_defs.o
12       poly.o
13       log.o
14       stats.o
15  #init maux_defs.o
16       _libmaux_malloc   malloc
17       _libmaux_log      log
```

Figure 13-9: Specification File

Briefly, here is what the specification file does. Lines 1 and 2 are comment lines. Lines 3 and 4 give the virtual addresses for the shared library text and data regions, respectively. Line 5 gives the pathname of the shared library on the target machine. The target shared library must be installed there for **a.out** files that use it to work correctly. Line 6 contains the **#branch** directive. Line 7 through 9 specify the branch table. They assign the functions **polyd()**, **logd()**, and **maux_stat()** to branch table entries 1, 2, and 3. Only external text symbols, such as C functions, should be placed in the branch table.

Line 10 contains the **#objects** directive. Lines 11 through 14 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, an addition of static data to **poly.o**, for example, would move external data symbols and break compatibility.

Line 15 contains the **#init** directive. Lines 16 and 17 give imported symbol information for the object file **maux_defs.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libmaux_malloc** will hold a pointer to **malloc**, and so on.

Building the Shared Library

Now, you have to compile the **.o** files as you would for any other library:

```
cc –c maux_defs.c poly.c log.c stats.c -G 0
```

Next, you need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib –s libmaux.sl –t libmaux_s –h libmaux_s.a
```

Presuming all of the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libmaux_s.a**, and the target library, **libmaux_s**. Before any **a.out** files built with **libmaux_s.a** can be executed, the target shared library **libmaux_s** will have to be moved to **/my/directory/libmaux_s** as specified in the specification file.

Using the Shared Library

To use the shared library with a file **x.c** that contains a reference to one or more of the routines in **libmaux**, you would issue the following command line:

```
cc x.c libmaux_s.a -lm -lc_s
```

This command line causes:

- the imported symbol pointer reference to **log** to be resolved from **libm** and
- the imported symbol pointer reference to **malloc** to be resolved with the shared version from **libc_s**.

Summary

This chapter describes the UNIX system shared libraries and explains how to use them. It also explains how to build your own shared libraries. Using any shared library almost always saves disk storage space, memory, and computer power; and running the UNIX system on smaller machines makes the efficient use of these resources increasingly important. Therefore, you should normally use a shared library whenever it's available.

Introduction

The IRIX™ operating system supports a powerful set of real-time programming features. You can use these features in combination to accurately time events, use signals as true interrupt routines, control allocation of real memory to the process, and provide for priority scheduling.

In addition, you can use the fully configurable kernel to install custom drivers when wanted. This provides for a range of response time and latency from very fast handling at device interrupt time to high priority dispatch of user processes to handle the event.

Advanced programming features (such as lightweight processes and mapped files) provide an environment in which you can construct tightly-coupled and server applications, assuring better response for real-time work.

A real-time system provides immediate response to specific external events. Thus, a programmer can schedule particular processes to run within a specified time limit after the occurrence of an event.

Optimal real-time response requires at least two processors: one to handle interrupts and other jobs, and one to service high-priority real-time jobs. A multiprocessor system can have deterministic real-time response if the unpredictable loads (such as interrupts) are handled on processors other than the processor running the real-time application.

The system scheduler has to do a fair amount of work to switch contexts from one process to another. The scheduling algorithm spends its time among three phases: saving the context of the current process, searching for the highest priority process it can run, and restoring the context of the select process. Real-time response can be greatly improved if context switches are eliminated. This can be accomplished with the shared resource group system call (*sproc*(2)) by putting multiple tasks into one process. One of these tasks is dedicated to handle real-time events. You can designate a particular CPU to run only the real-time task; thus the context of the process is always available in the real-time CPU maps and registers.

The rest of this chapter describes the real-time environment of IRIX multiprocessor systems including:

- real-time features
- optimal real-time environment
- real-time latency figures

Real-Time Features

This set of features refers to the programmer-visible features of the system. An example program (provided later in this chapter) shows how to incorporate these features into your program for optimal real-time response. Real-time features include:

- interval timers
- virtual memory control
- non-degrading priorities
- shared resource groups
- process blocking
- user synchronization primitives
- preemptable kernel

Interval Timers

In BSD4.2, Berkeley introduced a new facility called *interval timers*, often shortened to just "itimers." This facility provides microsecond-resolution for both timers and the time of day clock. An interval timer allows the user to specify both an offset from the current time (the delay), and the recurrence time (the interval). The timer will not fire until the delay has passed, and then will continue to fire at the end of each interval. See *getitimer*(2) for more information.

Three timers are provided, each of which delivers a separate signal to the process. The first is the real-time timer, which delivers the standard SIGALRM. The second is a process virtual time timer, which runs only when the process is running in user mode, delivering the signal SIGVTALRM. The third timer is the system virtual time timer, which runs when the process is in either user mode or the kernel is operating on behalf of the user. This timer delivers the signal SIGPROF.

Using the combination of itimers and reliable signals, it is possible to implement accurate handling of tasks at regular intervals. The resolution of the itimer is typically the same as that of the system clock, which is 10ms. A higher resolution itimer is possible at the cost of about 6-8% performance penalty on the system. However, this feature can be dynamically controlled at run time through the *ftimer*(1) facility, so that only systems needing such accuracy are penalized. A higher resolution itimer can deliver the signal SIGALARM to user program at the interval of about 833us.

The resolution of the high resolution itimer depends on the resolution of the system fast clock. This resolution can be adjusted by editing the FASTHZ parameter in the */usr/sysgen/mater.d/kernel* file. To automatically enable the system fast clock at boot time, set the variable, fastclock, in the kernel file, then run *lboot*(1) for the change to take effect. When the system fast clock is enabled, the resolution of the *gettimeofday*(2) system call is the same as that of the system fast clock.

Virtual Memory Control

In a paging system, it is always possible that the next reference a program makes, be it to text or data, may cause a page fault. Fixing up a page fault can take a long time, destroying any semblance of real-time behavior that a program may have. To address this problem, ranges of pages can be locked into real memory, thus avoiding page fault penalties.

IRIX provides *pin* and *unpin* capabilities for the programmer. When a range of addresses is *pinned* by the program, the kernel allocates memory to all pages in the range and locks them into core before returning to the user program. Conversely, *unpinning* pages undoes any previous locking. (See *mpin*(2) for more information.) This makes it possible for an application to manage its memory effectively, and insure that critical text and data (such as signal handlers) are always available. Note that it may not always be necessary to lock the entire segment; in fact it may not be possible on a system with limited memory.

To determine the entire size of text or data, refer to *end*(3C). Declared data may be moved around by the linker. If you want to selectively *pin* data, put all of it in a structure, and then *pin* the structure.

Note that dynamic growable space (stack and data) is not automatically locked. Use *sbrk(0)* to determine the end of your heap.

Non-Degrading Priorities

UNIX implements priority aging for processes. Thus, a process that is CPU bound has its priority lowered gradually as it runs. This insures that lower priority processes can eventually run and not be shut out of the CPU. This behavior is good in the original environments for which UNIX was designed: interactive users pounding on ASCII terminals. In real-time programming, the programmer often wishes to insure that a process will run immediately when an event occurs (the delivery of a timer signal, for instance). Specifying a high priority with the *nice*(2) system call is only a partial solution; even "niced" jobs age, which can mean a significant delay before a process gets to run.

To alleviate this problem, *non-degrading* (sometimes called fixed) priorities have been added to the IRIX kernel. With this facility, the program can specify a priority for a process which does not decrease over time, thus insuring that it maintains its priority order in the system. This makes it possible to guarantee that certain processes run in a timely manner, and that they can control the CPU as long as necessary to accomplish their tasks.

Non-degrading priorities are available in three bands:

- high priority band — above normal priority
- middle priority band — normal priority range (normal processes are in this band)
- low priority band — below normal priority

This gives the ability to support time-critical applications, server applications and batch processing, for example. See *schedctl*(2) for more information on how to schedule non-degrading priorities.

Non-degrading priorities above the normal band can be dangerous if misused. For instance, consider a process with the highest non-degrading priority in the system. If this process enters an infinite loop, then all other processes will be blocked from running, and it will be necessary to reset the machine to regain control. On a multiprocessor system, however, it is possible to dedicate a CPU to a particular task using non-degrading priorities.

Shared Resource Groups

At the lowest level, share groups implement a lightweight process facility. This means that a set of processes may share the same virtual address space, and thus have free access to shared data without copying overhead or expensive protocols. The processes in a share group are scheduled independently, and may run at either user or kernel level in parallel. These lightweight processes are supported in all configurations, be they single processor or multiprocessor.

IRIX also adds an innovation to this basic scheme: other resources may be selectively shared as well. For instance, file descriptors may be shared, meaning that if one process opens the descriptor, all other processes in the share group may access it as well. Other resources may be shared, such as the current directory or ulimit values.

Using this facility, it is possible to construct very tightly coupled multiprocessor applications with fast response times. For instance, if two processes share file descriptors, one can open a file and suffer the overhead of disk access and resource allocation while the other can maintain real-time response for display update. Once the descriptor is opened, the kernel will automatically propagate it across all processes in the share group, which is much cheaper than multiple opens. The real-time process will then have immediate access to data through the previously opened descriptor. Mapped files can be used in this instance for an even lower-overhead method for passing data.

Process Blocking

IRIX provides several new system calls for managing the run state of a process. A process may block another (after appropriate security checks) or itself, and may unblock any process. This blocking operation is implemented via a counter in the kernel, and thus is free of race conditions. The blocking operations have been tuned for high performance.

Such primitives find their use in many areas, such as user-level semaphore support. Real-time modeling and simulation programs can use such calls to implement preemptive schedulers at the user level. For instance, an operating system environment can be simulated using a shared process (sharing the virtual memory image) and blocking calls for process control. See *blockproc*(2) for more information.

User Synchronization Primitives

IRIX provides several extremely powerful shared memory models for programmers to use. To effectively use shared memory in real-time applications, the synchronization primitives used for access to that memory must be very fast. Invoking kernel primitives, such as System V messages or semaphores, is only acceptable in applications where latency isn't an issue.

Conveniently, there is a set of user-level primitives which allow very low latency synchronization between processes. If the program is running on a POWER Series™ system, then spinlocking automatically takes advantage of the built-in hardware synchronization bus. On other 4D systems, a fast software spinlock mechanism is used instead. Semaphores, which can block or unblock processes, use spinlocks for the fundamental control mechanism, and only interact with the kernel when putting a process to sleep or unblocking a waiting process.

These synchronization facilities also provide extensive tracing and metering support, which makes it much easier to debug and tune an application. See *uspsema*(3) and *usvsema*(3) for more information.

Preemptable Kernel

In standard UNIX, a process operating in system space (i.e., executing a system call) is not preemptable. A system call, even one from a low-priority user process, continues executing until it either blocks or runs to completion. This means that some additional latency may be introduced into the dispatch time for a process, depending on the activity on the system.

In the IRIX real-time environment, a kernel preemption checkpoint occurs whenever a high priority band, non-degrading type process becomes ready to run. Thus, the preemptive kernel reduces the delay so that system calls do not have to block or run to completion. This increases system response and further reduces latency.

At the present time, network processing under the IRIX real-time environment has no kernel preemption checkpoints embedded in it. Fortunately, on the multiprocessor POWER Series system, networking activities can be restricted to any one processor in the system. To lock networking activities to a processor, edit the */usr/sysgen/system* file and run *lboot*(1) for the changes to take effect. The format is:

NETWORKPROC: *cpu#* where *cpu#* is the number of the CPU on which you want all the networking activities to occur. For example:

NETWORKPROC: 1 designates the second processor to handle all networking activities. For real-time application, the real-time processor should be different from the networking processor.

Optimal Real-Time Environment

This section explains how to use the IRIX real-time features to set up optimal real-time response.

Establishing Multiprocessor Control

Multiprocessing eliminates unwanted interrupts and lower priority processes from competing with the real-time process for CPU cycles.

IRIX supports the *sysmp*(2) subcommands MP_RESTRICT and MP_MUSTRUN (see *sysmp*(2) for details on these subcommands). These system calls let you dedicate a processor on which to run the real-time program. These calls also let you restrict a particular processor to run only certain designated processes. You must be the superuser to use these commands. The MP_MUSTRUN command assigns the calling process to run only on the processor specified, except as required for communications with hardware devices. The MP_RESTRICT command restricts the processor specified from running any process except those assigned to it via a MP_MUSTRUN command, the *runon*(1) command, or because of hardware necessity.

Under IRIX, each processor in the system handles its own clock interrupt. The overhead of handling the clock interrupt is quite low for all the processors except for one special processor, the clock processor. It is selected by the MP_CLOCK (to specify a processor to handle the system clock) *sysmp* command. The real-time processor should not be the clock processor. Note that the program sample that follows uses various *sysmp* commands.

Locking Interrupts

On a single processor system, the time required for all interrupts to be serviced until no more interrupts are pending and process scheduling can proceed constitutes an unpredictable latency. Fortunately, on the multiprocessor POWER Series system, VME interrupt levels can be individually locked on to any processor in the system. For real-time application, you would move all the unwanted VME interrupt levels away from the real-time processor. To lock a particular VME interrupt level to a processor, edit the */usr/sysgen/system* file and run *lboot*(1) for the changes to take effect.

The format is:

IPL: *level cpu#* where *level* is the priority level (0 - 7, with 7 being the highest), and *cpu#* is the number of the CPU on which you want the VME interrupts of that level to occur. For example:

IPL: 4 1 designates VME interrupt priority level 4 on CPU number 1.

After editing the *system* file, you must run *lboot*(1) to reconfigure the system for the changes to take effect. See *lboot*(1) and *system*(4) for details.

A Real-Time Example

The following example shows how to set up a user program for real-time application and acquire system resources for optimal real-time processing. This program includes examples of the use of fine-grained memory locking, process blocking, shared process group, and reliable signals.

In this program, the user process is broken up into two threads executing on different processors. The real-time thread runs on a dedicated real- time processor, while the slave thread runs on a different processor.

The real-time thread execution:

- sets up the itimer to simulate a real-time device that keeps interrupting the real-time thread by periodically sending it a SIGALARM.
- does some cpu intensive calculation, trying not to sleep to avoid any context switching when it gets the itimer signal.
- on receipt of the itimer signal, wakes up the slave thread for additional I/O related processing.

```
   # include      <sys/types.h>
# include      <sys/time.h>
# include      <sys/schedctl.h>
# include      <sys/sysmp.h>
# include      <sys/pda.h>
# include      <signal.h>
# include      <setjmp.h>
# include      <stdio.h>
# include      <sys/prctl.h>

int            realtime_cpu = -1;
int            master_cpu = 0; /*default to cpu 0 to
                              * handle system clock*/
extern int     errno;

struct timeval lasttime;


int            npri;

struct {       /* structure we communicate through */
       int     ppid;  /* parent process ID */
       int     cpid;  /* child process ID */
       unsigned count;       /* counter, bumped by parent */
       unsigned nintr;       /* counter of itimer signals */
} comarea;

main(argc, argv)
   int                argc;
   char               *argv[];
{
   extern int         optind;
   extern char        *optarg;
   int                c;
   int                err;
   int                i;
   int                nprocs;
   int                catcher();
   register struct pda_stat *pstatus, *p;
   struct itimerval   itv;
   int asyncslave();
       /*
```

```
 * Parse arguments.
 */
while ((c = getopt(argc, argv, "r:f:")) != EOF) {
  switch (c) {
  case 'f':

    /*
     * Set a non-degrading or fixed priority
     * at the given value.
     */
      if ((npri = strtol(optarg, (char **) 0, 0)) <= 0) {
          err++;
          break;
      }

      /*
       * Figure out which band it is in.
       * Smaller values mean higher priorities.
       * These codes are here for illustration.
       */
      if (npri >= NDPHIMAX && npri <= NDPHIMIN) {
        /*
         * High priority (higher than all other
         * processes.)
         */
      }
      else if (npri >= NDPNORMMAX && npri <= NDPNORMMIN) {
        /*
         * Non-degrading in the normal UNIX priority
         * bands. This should give "constant response."
         */
      }
      else if (npri >= NDPLOMAX && npri <= NDPLOMIN) {
        /*
         * Lower than all other processes.  Suitable
         * for batch work, etc.
         */
      }
       else
          err++;
       break;
  case 'r':
      if ((realtime_cpu = strtol(optarg,
              (char **) 0, 0)) < 0)
```

```
                err++;
             break;
       case '?':
                err++;
             break;
       }
   }
   if (err) {
     fprintf(stderr, "usage: %s [-rcpu] [-fpri]0,
                       argv[0]);
     exit(1);
   }

  /*
   * Start the slave thread first.
   */
  comarea.ppid = getpid();
  comarea.cpid = sproc(asyncslave, PR_SALL, 0);

  /*
   * Pin the signal handlers down to improve signal
   * handling performance. We pass in a stack address
   * so we know which part of the stack to lock.
   */
  pinmem((char *) &itv);

  /*
   * Set the priority if requested.  If in the high
   * priority band, then you should always get control.
   * If in the middle band, depends on who else is
   * running.  If in the lower band, you only
   * get cycles if nothing else is running.  Note
   * that setting anon-degrading priority requires
   * super-user priviledges.
   */
  if (npri != 0)
     schedctl(NDPRI, 0, npri);

  /*
   * To dedicate a processor to handle interrupts
   * from a real-time device, use lboot to reconfig
   * the kernel and reboot the system (see system(4),
   * lboot(1m)). To run your process only on the
```

```
     * dedicated real-time processor,
     * then use the following system call.
     */
    if (realtime_cpu >= 0)
       if (sysmp(MP_MUSTRUN, realtime_cpu) < 0) {
         perror("Failed MP_MUSTRUN.
                       Resource not available!");
       }

    /*
     * To obtain maximum real-time processing power
     * out of the selectedprocessor, use the following
     * system call to kick all other processes
     * from the real-time processor.
     */
   if (realtime_cpu >= 0 && sysmp(MP_RESTRICT,
                        realtime_cpu) < 0) {
       perror("Failed MP_RESTRICT.
                        Resource not available!");
   }

    /*
     * move the handling of the system clock
     * to another cpu, if it is currently
     * handled by the real-time cpu. This will
     * increase the band-width of the real-time cpu.
     */
    nprocs = sysmp(MP_NPROCS);
    if (nprocs < 0) {
       perror("Failed MP_NPROCS. Fatal system error!");
       quit(-1);
    }
    pstatus = (struct pda_stat *)
       calloc(nprocs, sizeof(struct pda_stat));
    if (sysmp(MP_STAT, pstatus) < 0) {
       perror("Failed MP_STAT. Fatal system error!");
       quit(-1);
    }
    /*
     * Figure out which processor is currently
     * handling the system clock. If that processor
     * happens to be the real-time cpu then move
     * that functionality to another processor
     * (the master processor in this example).
     */
    for (i = 0, p = pstatus; i < nprocs; i++) {
       if ((p->p_flags & PDAF_CLOCK) &&
                      (i == realtime_cpu)) {
          if (sysmp(MP_CLOCK, master_cpu) < 0) {
               perror("Failed MP_CLOCK.
                      Fatal system error!");
               quit(-1);
          }
          break;
       }
       p++;
    }

    /*
     * Set up signal handling.  Initialize the alarm
     * signal to be "held", which means that
     * timer pops will be ignored until we
     * "release" the signal.
     */
    sigset(SIGALRM, catcher);
    sighold(SIGALRM);

    /*
     * Set up timer.  The interval is the time between
     * each successive timer pop. The value is the
     * initial value of the timer, which can be anything.
     * We will set the timer to start 10ms from now, and
     * keep interrupting every 10ms thereafter.
     */
    itv.it_interval.tv_sec = 0;
    itv.it_interval.tv_usec = 10000;
    itv.it_value = itv.it_interval;
    setitimer(ITIMER_REAL, &itv, (struct itimerval *) 0);

    /*
     * Get the starting time, and release
     * the interrupt signal.
     */
    gettimeofday(&lasttime, 0);
    sigrelse(SIGALRM);

    while (1) {
       /* simulate CPU intensive processing */
```

```
            comarea.count++;
        }
        /*NOTREACHED*/
}

/*
 * Slave thread
 * block waiting for parent to wake child up upon
 * receipt of the itimer signal.
 */
asyncslave()
{
    static struct timeval    last;
    static int lastintr;
    struct timeval    newtime;
    double            rt;
    int cdone();

    /*
     * Put us at the highest possible real-time
     * priority so we respond to events very quickly.
     */
    schedctl(NDPRI, 0, npri);

    last = lasttime;

    /* initialize semaphore count */
    setblockproccnt(comarea.cpid,0);

    /* catch INT signal to let parent know */
    sigset(SIGINT, cdone);

    for (;;) {
        /*
         * parent will wake us up
         */
        blockproc(comarea.cpid);
        /*
         * Get the time that we finished.
         */
        gettimeofday(&newtime, 0);

        /*
         * Turn the net time difference into a
         * floating point number so we can
         * reasonably deal with it.
         */
        rt = newtime.tv_sec - last.tv_sec;
        rt += (newtime.tv_usec - last.tv_usec) / 1000000;

        /*
         * Tell the user what the result was.
         */
        printf("Current rate of interrupts/sec = %.2f0,
                (comarea.nintr - lastintr)/rt);
        last = newtime;
        lastintr = comarea.nintr;
    }
    /*NOTREACHED*/
}

/*
** parent signal handler
*/
catcher()
{
    comarea.nintr++;
    /* wake up slave thread to output the results
     * every 100 signals
     */
    if ((comarea.nintr % 100) == 0)
        unblockproc(comarea.cpid);
}

/*
** code to lock critical memory in core lock
** stack, signal handler and the communication area
*/
pinmem(sbot)
   char        *sbot;
{

    /*
     * Pin down various pieces of critical
     * memory.  Start with the stack.
     * Allow about 2K for the stack (signal
     * handlers run on the same stack).
     */
```

```
        /* stack grows downward */
      mpin(sbot - (2*1024), 2*1024);

        /* pin data areas */
      mpin(&comarea, sizeof(comarea));

        /* pin text areas */
      mpin((char *) catcher, (int) pinmem - (int) catcher);
}

quit(code)
{
        kill(comarea.cpid, SIGINT);
        exit(code);
}

cdone()
{
        kill(comarea.ppid, SIGINT);
        exit(0);
}
```

Real-Time Latency

The following sections describe all the components that make up the process dispatch latency under IRIX. Process dispatch latency is the amount of time it takes after the occurrence of an interrupt before execution of the associated process begins. This latency is the sum of the interrupt latency, the interrupt service time, the kernel dispatch latency, and the context switch time, plus the unpredictable latencies introduced by any other pending interrupts at the time.

The Components of Process Dispatch Latency

The following components constitute the process dispatch latency for a real-time process under IRIX. Note that the figures are subject to change.

1. **Interrupt latency.** This is the maximum amount of time it takes after the occurrence of an interrupt before execution of the appropriate interrupt handler. Certain critical sections of the kernel have to be executed without allowing any interrupts to be handled. If a real-time event interrupts the processor when the kernel is executing one of these critical sections, it will have to wait. This wait (called the interrupt latency) is measured to be always less than 650 us.
2. **Interrupt service time.** The real-time interrupt handler performs a number of tasks, it wakes up sleeping processes waiting for the transfer to complete, then clears the interrupt and returns. The time required to do all of this is called the interrupt service time. This number depends on the number of tasks that the real-time interrupt handler has to perform (thus no number is supplied).
3. **Time to service other pending interrupts.** Once the system finishes servicing the real-time event, there are other factors that may prevent the system from dispatching the real-time process immediately to respond to the real-time event. One of these factors is other unrelated pending interrupts. The time required for all interrupts to be serviced until no more interrupts are pending and process scheduling can proceed constitutes an additional and unpredictable latency. On a single processor system, this latency depends on how much load there is on the system. On the multi-processor POWER Series system, all unwanted VME interrupt levels can be rerouted away from the real-time processor (including local SCSI and networking). In this case, the latency due to other pending interrupts is no longer a factor on the real-time processor.

4. **Time to service the clock interrupt.** The clock interrupt also constitutes an additional latency. The maximum clock interrupt service time is recorded to be less than 1.1ms for the clock processor and less than 120us for the other processors. Therefore, by not configuring the real-time processor as the clock processor, this latency is less than 120us.

5. **Kernel dispatch latency.** This is the delay between when a real-time process is ready to run again and when it is actually dispatched. In standard UNIX, a process operating in the kernel is not preemptable. A system call continues executing until it either blocks or runs to completion. This can add significant delay before the real-time process can be dispatched. Measurements show that if a real-time processor does not handle any interrupts other than the real-time event and the clock interrupt, critical sections of the user program are locked in core and execution in the kernel on behalf of the user is limited to be of non graphics nature, then this latency never exceeds 775us. This latency also includes the clock handling time.

6. **Context switch time.** Once the real-time process becomes ready to run again, and the executing lower-priority process is not executing a system call, the kernel will switch out the lower-priority process and switch in the real-time process. The time required to switch processes is termed the context switch time.

   This time is derived by having two processes running on the same processor, communicating to each other using a pipe. The context switch time is the delay between writing to a pipe and picking up the data (by the receiving process) on the other end of the pipe. This delay is measured to be less than 87 microseconds.

Maximum Process Dispatch Latency

On the multiprocessor POWER Series system, by following the guidelines in configuring the system for real-time application the maximum process dispatch latency is just the sum of the interrupt latency, kernel dispatch latency, the context switch time plus the interrupt service time. Excluding the interrupt service time, this maximum latency is always less than 1.5ms. The interrupt service time varies depending on the tasks involved in handling the real-time event.

Summary

This chapter described the IRIX operating system real-time and multiprocessor programming tools, which provide the ability to tune a multiprocessor system to the application. Real-time programming provides ways to reduce or avoid kernel overhead, which is the usual weakness of most implementations. The real-time and multiprocessor programming tools provide a satisfying base for real-time and quick response systems.

Index to Utilities

Throughout the text of this guide, commands are discussed without identifying the package to which the command belongs. The assumption has been that all command packages are present on the machine on which you are working.

If some commands seem to produce only a `not found` message on your computer, it may be that the package to which the command belongs has not been installed. If that happens, check with the administrator of your system.

■ Basic Networking Utilities

ct	ct(1C)
cu	cu(1C)
uutry	uutry(1M)
uucheck	uucheck(1M)
uucico	uucico(1M)
uucleanup	uucleanup(1M)
uucp	uucp(1C)
uugetty	uugetty(1M)
uulog	uucp(1C)
uuname	uucp(1C)
uupick	uuto(1C)
uusched	uusched(1M)
uustat	uustat(1C)
uuto	uuto(1C)
uux	uux(1C)
uuxqt	uuxqt(1M)

■ C Programming Language Utilities

cc	cc(1)
cpp	cpp(1)
list	list(1)

■ Advanced C Utilities

cb	cb(1)
cflow	cflow(1)
ctrace	ctrace(1)
cxref	cxref(1)
lint	lint(1)
regcmp	regcmp(1)

■ Directory and File Management Utilities

ar	ar(1)
awk	awk(1)
bdiff	bdiff(1)
bfs	bfs(1)
col	col(1)
comm	comm(1)
csplit	csplit(1)
cut	cut(1)
diff3	diff3(1)
dircmp	dircmp(1)
egrep	egrep(1)
fgrep	fgrep(1)
find	find(1)
join	join(1)
newform	newform(1)
nl	nl(1)
od	od(1)
pack	pack(1)
paste	paste(1)
pcat	pack(1)
pg	pg(1)
sdiff	sdiff(1)
split	split(1)
sum	sum(1)
tail	tail(1)
touch	touch(1)
tr	tr(1)
uniq	uniq(1)
unpack	pack(1)

■ Editing Utilities

edit	edit(1)
ex	ex(1)
vi	vi(1)

■ Essential Utilities

brc	brc(1M)
cat	cat(1)
cd	cd(1)
checkall	fsck(1M)
chgrp	chown(1)
chmod	chmod(1)
chown	chown(1)

ckbupscd	ckbupscd(1M)
clri	clri(1M)
cmp	cmp(1)
cp	cp(1)
cpio	cpio(1)
cron	cron(1M)
date	date(1)
dd	dd(1M)
devinfo	devinfo(1M)
devnm	devnm(1M)
df	df(1M)
diff	diff(1)
du	du(1M)
echo	echo(1)
ed	ed(1)
expr	expr(1)
false	true(1)
file	file(1)
fsck	fsck(1M)
fsstat	fsstat(1M)
fstyp	fstyp(1M)
getopt	getopt(1)
getoptcvt	getoptcvt(1)
getopts	getopts(1)
getty	getty(1M)
grep	grep(1)
id	id(1M)
init	init(1M)
kill	kill(1)
killall	killall(1M)
labelit	labelit(1M)
ln	cp(1)
login	login(1)
ls	ls(1)
machid	machid(1)
mail	mail(1)
mesg	mesg(1)
mkdir	mkdir(1)
makehdfsys	makehdfsys(1M)
mkfs	mkfs(1M)
mkmenus	mkmenus(1)
mknod	mknod(1M)
mkunix	mkunix(1M)
mount	mount(1M)

mountall .... mountall(1M)
mv .... cp(1)
newgrp .... newgrp(1M)
news .... news(1)
passwd .... passwd(1)
pdp11 .... machid(1)
powerdown .... powerdown(1M)
pr .... pr(1)
prtvtoc .... prtvtoc(1M)
ps .... ps(1)
pwd .... pwd(1)
rc0 .... rc0(1M)
rc2 .... rc2(1M)
red .... ed(1)
rm .... rm(1)
rmail .... mail(1)
rmdir .... rm(1)
rsh .... sh(1)
sed .... sed(1)
setmnt .... setmnt(1M)
setup .... setup(1)
sh .... sh(1)
shutdown .... shutdown(1M)
sleep .... sleep(1)
sort .... sort(1)
stty .... stty(1)
su .... su(1M)
sync .... sync(1M)
sysadm .... sysadm(1)
tee .... tee(1)
test .... test(1)
touch .... touch(1)
true .... true(1)
u3b2 .... machid(1)
umask .... umask(1)
umount .... mount(1M)
umountall .... mountall(1M)
uname .... uname(1)
wait .... wait(1)
wall .... wall(1)
wc .... wc(1)
who .... who(1)
write .... write(1)

■ **FORTRAN Programming Language Utilities**

asa .... asa(1)
f77 .... f77(1)
fsplit .... fsplit(1)
ratfor .... ratfor(1)

■ **Inter-process Communications Utilities**

ipcrm .... ipcrm(1)
ipcs .... ipcs(1)

■ **Line Printer Spooling Utilities**

accept .... accept(1M)
cancel .... lp(1)
disable .... enable(1)
enable .... enable(1)
lp .... lp(1)
lpadmin .... lpadmin(1M)
lpsched .... lpsched(1M)
lpstat .... lpstat(1)
reject .... accept(1M)

■ **Performance Measurement Utilities**

profiler .... profiler(1M)
sadp .... sadp(1M)
sar .... sar(1)
sar .... sar(1M)
timex .... timex(1)

■ **Security Administration Utilities**

crypt .... crypt(1)
makekey .... makekey(1)

■ **Software Generation Utilities**

ar .... ar(1)
as .... as(1)
dis .... dis(1)
dump .... dump(1)
ld .... ld(1)
lorder .... lorder(1)
m4 .... m4(1)
mkshlib .... mkshlib(1)
nm .... nm(1)
size .... size(1)
strip .... strip(1)

tsort	tsort(1)

Extended Software Generation Utilities

lex	lex(1)
install	install(1M)
make	make(1)
prof	prof(1)
yacc	yacc(1)

Source Code Control System Utilities

admin	admin(1)
cdc	cdc(1)
comb	comb(1)
delta	delta(1)
get	get(1)
help	help(1)
prs	prs(1)
rmdel	rmdel(1)
sact	sact(1)
sccsdiff	sccsdiff(1)
unget	unget(1)
val	val(1)
vc	vc(1)
what	what(1)

Spell Utilities

deroff	deroff(1)
hashcheck	spell(1)
hashmake	spell(1)
spell	spell(1)
spellin	spell(1)

System Administration Utilities

chroot	chroot(1M)
fuser	fuser(1M)
link	link(1M)
mvdir	mvdir(1M)
ncheck	ncheck(1M)
pwck	pwck(1M)
swap	swap(1M)
uadmin	uadmin(1M)
whodo	whodo(1M)

■ Terminal Information Utilities

captoinfo .... captoinfo(1M)
infocmp .... infocmp(1M)
tic .... tic(1M)
tput .... tput(1)

■ User Environment Utilities

at .... at(1)
banner .... banner(1)
basename .... basename(1)
batch .... at(1)
bc .... bc(1)
cal .... cal(1)
calendar .... calendar(1)
crontab .... crontab(1)
dc .... dc(1)
dirname .... basename(1)
env .... env(1)
factor .... factor(1)
line .... line(1)
logname .... logname(1)
nice .... nice(1)
nohup .... nohup(1)
tabs .... tabs(1)
time .... time(1)
tty .... tty(1)
u3b .... machid(1)
u3b5 .... machid(1)
units .... units(1)
vax .... machid(1)
xargs .... xargs(1)

Glossary

Ada	Named after the Countess of Lovelace, the nineteenth century mathematician and computer pioneer, Ada is a high-level general-purpose programming language developed under the sponsorship of the U.S. Department of Defense. Ada was developed to provide consistency among programs originating in different branches of the military. Ada features include packages that make data objects visible only to the modules that need them, task objects that facilitate parallel processing, and an exception handling mechanism that encourages well-structured error processing.
ANSI standard	ANSI is the acronym for the American National Standards Institute. ANSI establishes guidelines in the computing industry, from the definition of ASCII to the determination of overall datacom system performance. ANSI standards have been established for both the Ada and FORTRAN programming languages, and a standard for C has been proposed.
a.out file	**a.out** is the default file name used by the link editor when it outputs a successfully compiled, executable file. **a.out** contains object files that are combined to create a complete working program. Object file format is described in Chapter 11, "The Common Object File Format," and in **a.out**(4) in the *IRIS-4D Programmer's Reference Manual.*
application program	An application program is a working program in a system. Such programs are usually unique to one type of users' work, although some application programs can be used in a variety of business situations. An accounting application, for example, may well be applicable to many different businesses.
archive	An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command **ar**(1) collects data for use as a library.
argument	An argument is additional information that is passed to a command or a function. On a command line, an argument is a character string or number that follows the command name and is separated from it by a space. There are two types of command-line arguments:

	options and operands. Options are immediately preceded by a minus sign (–) and change the execution or output of the command. Some options can themselves take arguments. Operands are preceded by a space and specify files or directories that will be operated on by the command. For example, in the command **pr –t –h Heading file** all of the elements after the **pr** are arguments. **–t** and **–h** are options, **Heading** is an argument to the **–h** option, and **file** is an operand. For a function, arguments are enclosed within a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if more than two are present they are separated by commas and the whole list enclosed by the parentheses. The formal definition of a function, such as might be found on a page in Section 3 of the *IRIS-4D Programmer's Reference Manual*, describes the number and data type of argument(s) expected by the function.
ASCII	ASCII is an acronym for American Standard Code for Information Interchange, a standard for data representation that is followed in the UNIX system. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lower-case letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is one byte long.
assembler	The assembler is a translating program that accepts instructions written in the assembly language of the computer and translates them into the binary representation of machine instructions. In many cases, the assembly language instructions map 1 to 1 with the binary machine instructions.
assembly language	A programming language that uses the instruction set that applies to a particular computer.
branch table	A branch table is an implementation technique for fixing the addresses of text symbols, without forfeiting the ability to update code. Instead of being directly associated with function code, text symbols label jump instructions that transfer control to the real code. Branch table

	addresses do not change, even when one changes the code of a routine. Jump table is another name for branch table.
buffer	A buffer is a storage space in computer memory where data are stored temporarily into convenient units for system operations. Buffers are often used by programs, such as editors, that access and alter text or data frequently. When you edit a file, a copy of its contents are read into a buffer where you make changes to the text. For the changes to become part of the permanent file, you must write the buffer contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.
byte	A byte is a unit of storage in the computer. On many UNIX systems, a byte is eight bits (binary digits), the equivalent of one character of text.
byte order	Byte order refers to the order in which data are stored in computer memory.
C	The C programming language is a general-purpose programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both high-level and low-level tasks. Although it has been called a system programming language, because it is useful for writing operating systems, it has been used equally effectively to write major numerical, text-processing, and database programs. The C programming language was designed for and implemented on the UNIX system; however, the language is not limited to any one operating system or machine.
C compiler	The C compiler converts C programs into assembly language programs that are eventually translated into object files by the assembler.
C preprocessor	The C preprocessor is a component of the C Compilation System. In C source code, statements preceded with a pound sign (#) are directives to the preprocessor. Command line options of the **cc**(1) command may also be used to control the actions of the preprocessor. The main work of the preprocessor is to perform file inclusions and macro substitution.

CCS	CCS is an acronym for C Compilation System, which is a set of programming language utilities used to produce object code from C source code. The major components of a C Compilation System are a C preprocessor, C compiler, assembler, and link editor. The C preprocessor accepts C source code as input, performs any preprocessing required, then passes the processed code to the C compiler, which produces assembly language code that it passes to the assembler. The assembler in turn produces object code that can be linked to other object files by the link editor. The object files produced are in the Common Object File Format (COFF). Other components of CCS include a symbolic debugger, an optimizer that makes the code produced as efficient as possible, productivity tools, tools used to read and manipulate object files, and libraries that provide runtime support, access to system calls, input/output, string manipulation, mathematical functions, and other code processing functions.
COFF	COFF is an acronym for Common Object File Format. COFF refers to the format of the output file produced on some UNIX systems by the assembler and the link editor. This format is also used by other operating systems. The following are some of its key features: ☐ Applications may 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. ☐ Users may make some modifications in the object file construction at compile time.
command	A command is the term commonly used to refer to an instruction that a user types at a computer terminal keyboard. It can be the name of a file that contains an executable program or a shell script that can be processed or executed by the computer on request. A command is composed of a word or string of letters and/or special characters that can continue for several (terminal) lines, up to 256 characters. A command name is sometimes used interchangeably with a program name.

command line	A command line is composed of the command name followed by any argument(s) required by the command or optionally included by the user. The manual page for a command includes a command line synopsis in a notation designed to show the correct way to type in a command, with or without options and arguments.
compiler	A compiler transforms the high-level language instructions in a program (the source code) into object code or assembly language. Assembly language code may then be passed to the assembler for further translation into machine instructions.
core	Core is a (mostly archaic) synonym for primary memory.
core file	A core file is an image of a terminated process saved for debugging. A core file is created under the name "core" in the current directory of the process when an abnormal event occurs resulting in the process' termination. A list of these events is found in the **signal**(2) manual page in section 2 of the *IRIS-4D Programmer's Reference Manual.*
core image	Core image is a copy of all the segments of a running or terminated program. The copy may exist in main storage, in the swap area. or in a core file.
curses	**curses**(3X) is a library of C routines that are designed to handle input, output, and other operations in screen management programs. The name **curses** comes from the cursor optimization that the routines provide. When a screen management program is run, cursor optimization minimizes the amount of time a cursor has to move about a screen to update its contents. The program refers to the **terminfo**(4) database at run time to obtain the information that it needs about the screen (terminal) being used. See **terminfo**(4) in the *IRIS-4D Programmer's Reference Manual.*
data symbol	A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. See text symbol.
database	A database is a bank of information on a particular subject or subjects. On-line databases are designed so that by using subject headings, key words, or key phrases you can search for, analyze, update, and print out data.

debug	Debugging is the process of locating and correcting errors in computer programs.
default	A default is the way a computer will perform a task in the absence of other instructions.
delimiter	A delimiter is an initial character that identifies the next character or character string as a particular kind of argument. Delimiters are typically used for option names on a command line; they identify the associated word as an option (or as a string of several options if the options are bundled). In the UNIX system command syntax, a minus sign (–) is most often the delimiter for option names, for example, –**s** or –**n**, although some commands also use a plus sign (+).
directory	A directory is a type of file used to group and organize other files or directories. A directory consists of entries that specify further files (including directories) and constitutes a node of the file system. A subdirectory is a directory that is pointed to by a directory one level above it in the file system organization. The **ls**(1) command is used to list the contents of a directory. When you first log onto the system, you are in your home directory ($HOME). You can move to another directory by using the **cd**(1) command and you can print the name of the current directory by using the **pwd**(1) command. You can also create new directories with the **mkdir**(1) command and remove empty directories with **rmdir**(1). A directory name is a string of characters that identifies a directory. It can be a simple directory name, the relative pathname or the full pathname of a directory.
dynamic linking	Dynamic linking refers to the ability to resolve symbolic references at run time. Systems that use dynamic linking can execute processes without resolving unused references. See static linking.
environment	An environment is a collection of resources used to support a function. In the UNIX system, the shell environment is composed of variables whose values define the way you interact with the system. For example, your environment includes your shell prompt string, specifics

	for backspace and erase characters, and commands for sending output from your terminal to the computer. An environment variable is a shell variable such as $HOME (which stands for your login directory) or $PATH (which is a list of directories the shell will search through for executable commands) that is part of your environment. When you log in, the system executes programs that create most of the environmental variables that you need for the commands to work. These variables come from **/etc/profile**, a file that defines a general working environment for all users when they log onto a system. In addition, you can define and set variables in your personal **.profile** file, which you create in your login directory to tailor your own working environment. You can also temporarily set variables at the shell level.
executable file	An executable file is a file that can be processed or executed by the computer without any further translation. That is, when you type in the file name, the commands in the file are executed. An object file that is ready to run (ready to be copied into the address space of a process to run as the code of that process) is an executable file. Files containing shell commands are also executable. A file may be given execute permission by using the **chmod**(1) command. In addition to being ready to run, a file in the UNIX system needs to have execute permission.
exit	A specific system call that causes the termination of a process. The **exit**(2) call will close any open files and clean up most other information and memory which was used by the process.
exit status: return code	An exit status or return code is a code number returned to the shell when a command is terminated that indicates the cause of termination.
expression	An expression is a mathematical or logical symbol or meaningful combination of symbols. See regular expression.
file	A file is an identifiable collection of information that, in the UNIX system, is a member of a file system. A file is known to the UNIX system as an inode plus the

	information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file may contain text, data, programs or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special or directory files.
file and record locking	The phrase "file and record locking" refers to software that protects records in a data file against the possibility of being changed by two users at the same time. Records (or the entire file) may be locked by one authorized user while changes are made. Other users are thus prevented from working with the same record until the changes are completed.
file descriptor	A file descriptor is a number assigned by the operating system to a file when the file is opened by a process. File descriptors 0, 1, and 2 are reserved; file descriptor 0 is reserved for standard input (**stdin**), 1 is reserved for standard output (**stdout**), and 2 is reserved for standard error output (**stderr**).
file system	A UNIX file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to the root, are branches. The collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.
filter	A filter is a program that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used as a data transformer in a pipeline. Filters are different from editors and other commands because filters do not change the contents of a file. Examples of filters are **grep**(1) and **tail**(1), which select and output part of the input; **sort**(1), which sorts the input; and **wc**(1), which counts the number of words, characters, and lines in the input. **sed**(1) and **awk**(1) are also filters but they are called programmable filters or data transformers because a program must be supplied as input in addition

	to the data to be transformed.
flag	A flag or option is used on a command line to signal a specific condition to a command or to request particular processing. UNIX system flags are usually indicated by a leading hyphen (–). The word option is sometimes used interchangeably with flag. Flag is also used as a verb to mean to point out or to draw attention to. See option.
fork	**fork**(2) is a system call that divides a new process into two, the parent and child processes, with separate, but initially identical, text, data, and stack segments. After the duplication, the child (created) process is given a return code of 0 and the parent is given the process id of the newly created child as the return code.
FORTRAN	FORTRAN is an acronym for FORmula TRANslator. FORTRAN is a high-level programming language originally designed for scientific and engineering calculations but now also widely adapted for many business uses.
function	A function is a task done by a computer. In most modern programming languages, programs are made up of functions and procedures which perform small parts of the total job to be done.
header file	A header file is used in programming and in document formatting. In a programming context, a header file is a file that usually contains shared data declarations that are to be copied into source programs as they are compiled. A header file includes symbolic names for constants, macro definitions, external variable references and inclusion of other header files. The name of a header file customarily ends with '.h' (dot-h). Similarly, in a document formatting context, header files contain general formatting macros that describe a common document type and can be used with many different document bodies.
high-level language	A high-level language is a computer programming language such as C or FORTRAN that uses symbols and command statements representing actions the computer is to perform, the exact steps for a machine to follow. A high-level language must be translated into machine language by a compilation system before a computer can execute it. A characteristic of a high-level language is that each statement usually translates into a series of

	machine language instructions. The low-level details of the computer's internal organization are left to the compilation system.
host machine	A host machine is the machine on which an **a.out** file is built.
interpreted language	An interpreted language is a high-level language that is not translated by a compilation system and stored in an executable object file. The statements of a program in an interpreted language are translated each time the program is executed.
Interprocess Communication	Interprocess Communication describes software that enables independent processes running at the same time, to exchange information through messages, semaphores, or shared memory.
interrupt	An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals that are generated by a hardware condition or a peripheral device indicating that a certain event has happened. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX programs by pressing the delete or break keys, by typing Control-d, or by using the **kill**(1) command.
I/O (Input/Output)	I/O is the process by which information enters (input) and leaves (output) the computer system.
kernel	The kernel (comprising 5 to 10 percent of the operating system software) is the basic resident software on which the UNIX system relies. It is responsible for most operating system functions. It schedules and manages the work done by the computer and maintains the file system. The kernel has its own text, data, and stack areas.
lexical analysis	Lexical analysis is the process by which a stream of characters (often comprising a source program) is subdivided into its elementary words and symbols (called tokens). The tokens include the reserved words of the language, its identifiers and constants, and special symbols such as =, :=, and ;. Lexical analysis enables you to recognize,

	for example, that the stream of characters 'print("hello, universe")' is to be analyzed into a series of tokens beginning with the word 'print' and not with, say, the string 'print("h.' In compilers, a lexical analyzer is often called by the compiler's syntactic analyzer or parser, which determines the statements of the program (that is, the proper arrangements of its tokens).
library	A library is an archive file that contains object code and/or files for programs that perform common tasks. The library provides a common source for object code, thus saving space by providing one copy of the code instead of requiring every program that wants to incorporate the functions in the code to have its own copy. The link editor may select functions and data as needed.
link editor	A link editor, or loader, collects and merges separately compiled object files by linking together object files and the libraries that are referenced into executable load modules. The result is an **a.out** file. Link editing may be done automatically when you use the compilation system to process your programs on the UNIX system, but you can also link edit previously compiled files by using the **ld**(1) command.
magic number	The magic number is contained in the header of an **a.out** file. It indicates what the type of the file is, whether shared or non-shared text, and on which processor the file is executable.
makefile	A makefile is a file that lists dependencies among the source code files of a software product and methods for updating them, usually by recompilation. The **make**(1) command uses the makefile to maintain self-consistent software.
manual page	A manual page, or "man page" in UNIX system jargon, is the repository for the detailed description of a command, a system call, subroutine or other UNIX system component.
null pointer	A null pointer is a C pointer with a value of 0.
object code	Object code is executable machine-language code produced from source code or from other object files by an assembler or a compilation system. An object file is a file of object code and associated data. An object file

that is ready to run is an executable file.

optimizer — An optimizer, an optional step in the compilation process, improves the efficiency of the assembly language code. The optimizer reduces the space used by and speeds the execution time of the code.

option — An option is an argument used in a command line to modify program output by modifying the execution of a command. An option is usually one character preceded by a hyphen (–). When you do not specify any options, the command will execute according to its default options. For example, in the command line

```
ls –a –l directory
```

–a and **–l** are the options that modify the **ls**(1) command to list all **directory** entries, including entries whose names begin with a period (.), in the long format (including permissions, size, and date).

parent process — A parent process occurs when a process is split into two, a parent process and a child process, with separate, but initially identical text, data, and stack segments.

parse — To parse is to analyze a sentence in order identify its components and to determine their grammatical relationship. In computer terminology the word has a similar meaning, but instead of sentences, program statements or commands are analyzed.

pathname — A pathname is a way of designating the exact location of a file in a file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is either a file or another directory. If the pathname begins with a slash, it is called a full pathname ; the initial slash means that the path begins at the **root** directory.

A pathname that does not begin with a slash is known as a relative pathname, meaning relative to the present working directory. A relative pathname may begin either with a directory name or with two dots followed by a slash (../). One that begins with a directory name indicates that the ultimate file or directory is below the present working directory in the hierarchy. One that begins with **../** indicates that the path first proceeds up the hierarchy; **../** is the parent of the present working directory.

permissions

Permissions are a means of defining a right to access a file or directory in the UNIX file system. Permissions are granted separately to you, the owner of the file or directory, your group, and all others. There are three basic permissions:

- ☐ Read permission (r) includes permission to cat, pg, lp, and **cp** a file.
- ☐ Write permission (w) is the permission to change a file.
- ☐ Execute permission (x) is the permission to run an executable file.

Permissions can be changed with the UNIX system **chmod**(1) command.

pipe

A pipe causes the output of one command to be used as the input for the next command so that the two run in sequence. You can do this by preceding each command after the first command with the pipe symbol (|), which indicates that the output from the process on the left should be routed to the process on the right. For example, in the command

who | wc –l,

the output from the **who**(1) command, which lists the users who are logged on to the system, is used as input for the word-count command, **wc**(1), with the **l** option. The result of this pipeline (succession of commands connected by pipes) is the number of people who are currently logged on to the system.

portable

Portability describes the degree of ease with which a program or a library can be moved or ported from one system to another. Portability is desirable because once a program is developed it is used on many systems. If the program writer must change the program in many different ways before it can be distributed to the other

	systems, time is wasted, and each modification increases the chances for an error.
preprocessor	Preprocessor is a generic name for a program that prepares an input file for another program. For example, **neqn**(1) and **tbl**(1) are preprocessors for **nroff**(1). **grap**(1) is a preprocessor for **pic**(1). **cpp**(1) is a preprocessor for the C compiler.
process	A process is a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current working directory, status of files, information recorded at login time, etc. Every time you type the name of a file that contains an executable program, you initiate a new process. Shell programs can cause the initiation of many processes because they can contain many command lines.
	The process id is a unique system-wide identification number that identifies an active process. The process status command, **ps**(1), prints the process ids of the processes that belong to you.
program	A program is a sequence of instructions or commands that cause the computer to perform a specific task, for example, changing text, making a calculation, or reporting on the status of the system. A subprogram is part of a larger program and can be compiled independently.
regular expression	A regular expression is a string of alphanumeric characters and special characters that describe a character string. It is a shorthand way of describing a pattern to be searched for in a file. The pattern-matching functions of **ed**(1) and **grep**(1), for example, use regular expressions.
routine	A routine is a discrete section of a program to accomplish a set of related tasks
semaphore	In the UNIX system, a semaphore is a sharable short unsigned integer maintained through a family of system calls which include calls for increasing the value of the semaphore, setting its value, and for blocking waiting for its value to reach some value. Semaphores are part of the UNIX system IPC facility.

shared memory	Shared memory is an IPC (interprocess communication) facility in which two or more processes can share the same data space.
shell	The shell is the UNIX system program—**sh**(1)—responsible for handling all interaction between you and the system. It is a command language interpreter that understands your commands and causes the computer to act on them. The shell also establishes the environment at your terminal. A shell normally is started for you as part of the login process. Three shells, the Bourne shell, the Korn shell and the C shell, are popular. The shell can also be used as a programming language to write procedures for a variety of tasks.
signal: signal number	A signal is a message that you send to processes or processes send to one another. The most common signals you might send to a process are ones that would cause the process to stop: for example, interrupt, quit, or kill. A signal sent by a running process is usually a sign of an an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.
source code	Source code is the programming-language version of a program. Before the computer can execute the program, the source code must be translated to machine language by a compilation system or an interpreter.
standard error	Standard error is an output stream from a program. It is normally used to convey error messages. In the UNIX system, the default case is to associate standard error with the user's terminal.
standard input	Standard input is an input stream to a program. In the UNIX system, the default case is to associate standard input with the user's terminal.
standard output	Standard output is an output stream from a program. In the UNIX system, the default case is to associate standard output with the user's terminal.
stdio: standard input-output	**stdio**(3S) is a collection of functions for formatted and character-by-character input-output at a higher level than the basic read, write, and open operations.

static linking	Static linking refers to the requirement that symbolic references be resolved before run time. See dynamic linking.
stream	A stream is an open file with buffering provided by the stdio package.
string	A string is a contiguous sequence of characters treated as a unit. Strings are normally bounded by white space(s), tab(s), or a character designated as a separator. A string value is a specified group of characters symbolized to the shell by a variable.
strip	**strip**(1) is a command that removes the symbol table and relocation bits from an executable file.
subroutine	A subroutine is a program that defines desired operations and may be used in another program to produce the desired operations. A subroutine can be arranged so that control may be transferred to it from a master routine and so that, at the conclusion of the subroutine, control reverts to the master routine. Such a subroutine is usually called a closed subroutine. A single routine may be simultaneously a subroutine with respect to another routine and a master routine with respect to a third.
symbol table	A symbol table describes information in an object file about the names and functions in that file. The symbol table and relocation bits are used by the link editor and by the debuggers.
symbol value	The value of a symbol, typically its virtual address, used to resolve references.
syntax	☐ Command syntax is the order in which command names, options, option arguments, and operands are put together to form a command on the command line. The command name is first, followed by options and operands. The order of the options and the operands varies from command to command. ☐ Language syntax is the set of rules that describe how the elements of a programming language may legally be used.

system call	A system call is a request by an active process for a service performed by the UNIX system kernel, such as I/O, process creation, etc. All system operations are allocated, initiated, monitored, manipulated, and terminated through system calls. System calls allow you to request the operating system to do some work that the program would not normally be able to do. For example, the **getuid**(2) system call allows you to inspect information that is not normally available since it resides in the operating system's address space.
target machine	A target machine is the machine on which an **a.out** file is run. While it may be the same machine on which the **a.out** file was produced, the term implies that it may be a different machine.
TCP/IP (Transmission Control Protocol/Internetwork Protocol)	TCP/IP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. It is the Department of Defense standard in packet networks.
terminal definition	A terminal definition is an entry in the **terminfo**(4) database that describes the characteristics of a terminal. See **terminfo**(4) and **curses**(3X) in the *IRIS-4D Programmer's Reference Manual.*
terminfo	☐ a group of routines within the curses library that handle certain terminal capabilities. For example, if your terminal has programmable function keys, you can use these routines to program the keys. ☐ a database containing the compiled descriptions of many terminals that can be used with **curses**(3X) screen management programs. These descriptions specify the capabilities of a terminal and how it performs various operations — for example, how many lines and columns it has and how its control characters are interpreted. A **curses**(3X) program refers to the database at run time to obtain the information that it needs about the terminal being used.

See **curses**(3X) in the *IRIS-4D Programmer's Reference Manual.* **terminfo**(4) routines can be used in shell programs, as well as C programs.

text symbol	A text symbol is a symbol, usually a function name, that is defined in the **.text** portion of an **a.out** file.
tool	A tool is a program, or package of programs, that performs a given task.
trap	A trap is a condition caused by an error where a process state transition occurs and a signal is sent to the currently running process.

UNIX operating system

The UNIX operating system is a general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T. An operating system is the software on the computer under which all other software runs. The UNIX operating system has two basic parts:

- ☐ The kernel is the program that is responsible for most operating system functions. It schedules and manages all the work done by the computer and maintains the file system. It is always running and is invisible to users.
- ☐ The shell is the program responsible for handling all interaction between users and the computer. It includes a powerful command language called shell language.

The utility programs or UNIX system commands are executed using the shell, and allow users to communicate with each other, edit and manipulate files, and write and execute programs in several programming languages.

userid	A userid is an integer value, usually associated with a login name, used by the system to identify owners of files and directories. The userid of a process becomes the owner of files created by the process and descendent (forked) processes.
utility	A utility is a standard, permanently available program used to perform routine functions or to assist a programmer in the diagnosis of hardware and software errors, for

	example, a loader, editor, debugging, or diagnostics package.
variable	☐ A variable in a computer program is an object whose value may change during the execution of the program, or from one execution to the next. ☐ A variable in the shell is a name representing a string of characters (a string value). ☐ A variable normally set only on a command line is called a parameter (positional parameter and keyword parameter). ☐ A variable may be simply a name to which the user (user-defined variable) or the shell itself may assign string values.
white space	White space is one or more spaces, tabs, or newline characters. White space is normally used to separate strings of characters, and is required to separate the command from its arguments on a command line.
word	A word is a unit of storage in a computer that is composed of bytes of information.