

**UNISYS**

**System V Operating  
System**

**Programmer's  
Guide**

**Volume 2**

Unisys is a trademark of Unisys Corporation.

January 1988

Priced Item

Printed in U S America  
UP-13690

This document is intended for software releases based on AT&T Release 3 of UNIX System V or a subsequent release of the System unless otherwise indicated.

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

**NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT.** Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

PDP and VT100 are trademarks of Digital Equipment Corporation. Teletype is a registered trademark of AT&T. UNIX is a registered trademark of AT&T in the USA and other countries.

Portions of this material are copyrighted © by  
AT&T Technologies  
and are reprinted with their permission.

---

# Chapter 9: Interprocess Communication

Introduction	9-1
Messages	9-2
Getting Message Queues	9-8
Using <b>msgget</b>	9-8
Example Program	9-13
Controlling Message Queues	9-17
Using <b>msgctl</b>	9-17
Example Program	9-19
Operations for Messages	9-26
Using <b>msgop</b>	9-26
Example Program	9-28
Semaphores	9-41
Using Semaphores	9-43
Getting Semaphores	9-47
Using <b>semget</b>	9-47
Example Program	9-52
Controlling Semaphores	9-56
Using <b>semctl</b>	9-57
Example Program	9-59
Operations on Semaphores	9-72
Using <b>semop</b>	9-72
Example Program	9-74
Shared Memory	9-81
Using Shared Memory	9-82
Getting Shared Memory Segments	9-86
Using <b>shmget</b>	9-86

## Table of Contents

---

Example Program	9-81
Controlling Shared Memory	9-95
Using <b>shmctl</b>	9-95
Example Program	9-97
Operations for Shared Memory	9-107
Using <b>shmop</b>	9-107
Example Program	9-109

---

# Introduction

The UNIX system supports three types of Inter-Process Communication (IPC):

- messages
- semaphores
- shared memory

This chapter describes the system calls for each type of IPC.

Included in the chapter are several example programs that show the use of the IPC system calls. All of the example programs have been compiled and run on an AT&T 3B2 Computer.

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 "nonblocking 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

## Messages

---

- 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 */
    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 9-1.

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 9-1: **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 *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 9-2 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 9-2: 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 9-3 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 9-3: 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
<b>msgflg</b>	=	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:

```

msgqid = msgget (key, (IPC_CREAT | 0400));
msgqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
    
```

As specified by the **msgget(2)** page in the *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:



<b>msgflg</b>	====	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 *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 9-4) 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 *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.

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

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/msg.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key; /*declare as long integer*/
13     int opperm, flags;
14     int msqid, opperm_flags;
15     /*Enter the desired key*/
16     printf("Enter the desired key in hex = ");
17     scanf("%x", &key);

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

Figure 9-4: **msgget()** System Call Example (Sheet 1 of 3)

```
23      /*Set the desired flags.*/
24      printf("\nEnter corresponding number to\n");
25      printf("set the desired flags:\n");
26      printf("No flags          = 0\n");
27      printf("IPC_CREAT          = 1\n");
28      printf("IPC_EXCL           = 2\n");
29      printf("IPC_CREAT and IPC_EXCL = 3\n");
30      printf("          Flags      = ");

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

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

36      /*Incorporate the control fields (flags) with
37       the operation permissions*/
38      switch (flags)
39      {
40      case 0:      /*No flags are to be set.*/
41          opperm_flags = (opperm | 0);
42          break;
43      case 1:      /*Set the IPC_CREAT flag.*/
44          opperm_flags = (opperm | IPC_CREAT);
45          break;
46      case 2:      /*Set the IPC_EXCL flag.*/
47          opperm_flags = (opperm | IPC_EXCL);
48          break;
49      case 3:      /*Set IPC_CREAT & IPC_EXCL flags.*/
50          opperm_flags = (opperm | IPC_CREAT |
51                         IPC_EXCL);
52      }
```

Figure 9-4: `msgget()` System Call Example (Sheet 2 of 3)

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

54     /*Perform following if call unsuccessful.*/
55     if(msgqid == -1)
56     {
57         printf ("\nThe msgget system call failed!\n");
58         printf ("The error number = %d\n", errno);
59     }

60     /*Return msgqid upon successful completion*/
61     else
62         printf ("\nThe msgqid = %d\n", msgqid);
63     exit(0);
64 }
```

Figure 9-4: `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 *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 9-5) 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 *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:

<b>uid</b>	used to store the IPC_SET value for the effective user identification
<b>gid</b>	used to store the IPC_SET value for the effective group identification
<b>mode</b>	used to store the IPC_SET value for the operation permissions
<b>bytes</b>	used to store the IPC_SET value for the number of bytes in the message queue ( <b>msg_qbytes</b> )
<b>rtrn</b>	used to store the return integer value from the system call

<b>msqid</b>	used to store and pass the message queue identifier to the system call
<b>command</b>	used to store the code for the desired control command so that subsequent processing can be performed on it
<b>choice</b>	used to determine which member is to be changed for the IPC_SET control command
<b>msqid_ds</b>	used to receive the specified message queue identifier's data structure when an IPC_STAT control command is performed
<b>*buf</b>	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.

```
1  /*This is a program to illustrate
2  **the message control, msgctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode, bytes;
15     int rtn, msqid, command, choice;
16     struct msqid_ds msqid_ds, *buf;
17     buf = &msqid_ds;

18     /*Get the msqid, and command.*/
19     printf("Enter the msqid = ");
20     scanf("%d", &msqid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");
23     printf("IPC_STAT   = 1\n");
24     printf("IPC_SET    = 2\n");
25     printf("IPC_RMID  = 3\n");
26     printf("Entry     = ");
27     scanf("%d", &command);
```

Figure 9-5: msgctl() System Call Example (Sheet 1 of 4)

---

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

31      switch (command)
32      {
33      case 1:      /*Use msgctl() to duplicate
34                  the data structure for
35                  msqid in the msqid_ds area pointed
36                  to by buf and then print it out.*/
37                  rtn = msgctl(msqid, IPC_STAT,
38                              buf);
39                  printf ("\nThe USER ID = %d\n",
40                          buf->msg_perm.uid);
41                  printf ("The GROUP ID = %d\n",
42                          buf->msg_perm.gid);
43                  printf ("Operation permissions = 0%o\n",
44                          buf->msg_perm.mode);
45                  printf ("The msg_qbytes = %d\n",
46                          buf->msg_qbytes);
47                  break;
48      case 2:      /*Select and change the desired
49                  member(s) of the data structure.*/
50                  /*Get the original data for this msqid
51                  data structure first.*/
52                  rtn = msgctl(msqid, IPC_STAT, buf);
53                  printf("\nEnter the number for the\n");
54                  printf("member to be changed:\n");
55                  printf("msg_perm.uid   = 1\n");
56                  printf("msg_perm.gid   = 2\n");
57                  printf("msg_perm.mode  = 3\n");
58                  printf("msg_qbytes   = 4\n");
59                  printf("Entry       = ");
```

Figure 9-5: msgctl() System Call Example (Sheet 2 of 4)

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

66     switch(choice){
67     case 1:
68         printf("\nEnter USER ID = ");
69         scanf ("%d", &uid);
70         buf->msg_perm.uid = uid;
71         printf("\nUSER ID = %d\n",
72             buf->msg_perm.uid);
73         break;
74     case 2:
75         printf("\nEnter GROUP ID = ");
76         scanf("%d", &gid);
77         buf->msg_perm.gid = gid;
78         printf("\nGROUP ID = %d\n",
79             buf->msg_perm.gid);
80         break;
81     case 3:
82         printf("\nEnter MODE = ");
83         scanf("%o", &mode);
84         buf->msg_perm.mode = mode;
85         printf("\nMODE = 0%o\n",
86             buf->msg_perm.mode);
87         break;
```

Figure 9-5: msgctl() System Call Example (Sheet 3 of 4)

```
88         case 4:
89             printf("\nEnter msq_bytes = ");
90             scanf("%d", &bytes);
91             buf->msg_qbytes = bytes;
92             printf("\nmsg_qbytes = %d\n",
93                 buf->msg_qbytes);
94             break;
95         }

96         /*Do the change.*/
97         rtn = msgctl(msqid, IPC_SET,
98             buf);
99         break;

100        case 3:    /*Remove the msqid along with its
101                   associated message queue
102                   and data structure.*/
103            rtn = msgctl(msqid, IPC_RMID, NULL);
104        }
105        /*Perform following if call unsuccessful.*/
106        if(rtn == -1)
107        {
108            printf ("\nThe msgctl system call failed!\n");
109            printf ("The error number = %d\n", errno);
110        }
111        /*Return msqid upon successful completion*/
112        else
113            printf ("\nMsgctl successful for
114                msqid = %d\n", msqid);
115        exit (0);
116    }
```

Figure 9-5: `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 *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 9-6) 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 *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 **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message size (**MSGMAX**) for the 3B2 Computer where in **msgbuf** it is set to one (1) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.
- rcvbuf** used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)
- \*msgp** used as a pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers
- i** used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd()** system call; it is also used as a counter to output the received message for the **msgrcv()** system call

## Messages

---

<b>c</b>	used to receive the input character from the <b>getchar()</b> function (line 50)
<b>flag</b>	used to store the code of <b>IPC_NOWAIT</b> for the <b>msgsnd()</b> system call (line 61)
<b>flags</b>	used to store the code of the <b>IPC_NOWAIT</b> or <b>MSG_NOERROR</b> flags for the <b>msgrcv()</b> system call (line 117)
<b>choice</b>	used to store the code for sending or receiving (line 30)
<b>rtrn</b>	used to store the return values from all system calls
<b>msqid</b>	used to store and pass the desired message queue identifier for both system calls
<b>msgsz</b>	used to store and pass the <b>size</b> of the message to be sent or received
<b>msgflg</b>	used to pass the value of <b>flag</b> for sending or the value of <b>flags</b> for receiving
<b>msgtyp</b>	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.

```
1  /*This is a program to illustrate
2  **the message operations, msgop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

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

14 /*Start of main C language program*/
15 main()
16 {
17     extern int errno;
18     int i, c, flag, flags, choice;
19     int rtrn, msqid, msgsz, msgflg;
20     long mtype, msgtyp;
21     struct msqid_ds msqid_ds, *buf;
22     buf = &msqid_ds;
```

Figure 9-6: msgop() System Call Example (Sheet 1 of 7)

---

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

31     if(choice == 1) /*Send a message.*/
32     {
33         msgp = &sndbuf; /*Point to user send
                          structure.*/

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

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

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

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

Figure 9-6: `msgop()` System Call Example (Sheet 2 of 7)

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

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

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

57      /*Set the IPC_NOWAIT flag if
58      desired.*/
59      printf("\nEnter a 1 if you want the\n");
60      printf("the IPC_NOWAIT flag set: ");
61      scanf("%d", &flag);
62      if(flag == 1)
63          msgflg |= IPC_NOWAIT;
64      else
65          msgflg = 0;

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

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

Figure 9-6: `msgop()` System Call Example (Sheet 3 of 7)



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

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

82         /*Print out the affected members.*/

83         /*Print the incremented number of
84         messages on the queue.*/
85         printf("\nThe msg_qnum = %d\n",
86         buf->msg_qnum);
87         /*Print process id of last sender.*/
88         printf("The msg_lspid = %d\n",
89         buf->msg_lspid);
90         /*Print the last send time.*/
91         printf("The msg_stime = %d\n",
92         buf->msg_stime);
93     }
94 }

95     if(choice == 2) /*Receive a message.*/
96     {
97         /*Initialize the message pointer
98         to the receive buffer.*/
99         msgp = &rcvbuf;

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

Figure 9-6: `msgop()` System Call Example (Sheet 4 of 7)

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

108     /*Configure the control flags for the
109         desired actions.*/
110     printf("\nEnter the corresponding code\n");
111     printf("to select the desired flags: \n");
112     printf("No flags                = 0\n");
113     printf("MSG_NOERROR                 = 1\n");
114     printf("IPC_NOWAIT                    = 2\n");
115     printf("MSG_NOERROR and IPC_NOWAIT = 3\n");
116     printf("Flags                          = ");
117     scanf("%d", &flags);

118     switch(flags) {
119         /*Set msgflg by ORing it with the
120             appropriate flags (constants).*/
121     case 0:
122         msgflg = 0;
123         break;
124     case 1:
125         msgflg |= MSG_NOERROR;
126         break;
127     case 2:
128         msgflg |= IPC_NOWAIT;
129         break;
130     case 3:
131         msgflg |= MSG_NOERROR | IPC_NOWAIT;
132         break;
133     }
```

Figure 9-6: `msgop()` System Call Example (Sheet 5 of 7)

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

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

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

145      if(rtrn == -1) {
146          printf("\nMsgrcv failed. ");
147          printf("Error = %d\n", errno);
148      }
149      else {
150          printf ("\nMsgctl was successful\n");
151          printf("for msqid = %d\n",
152              msqid);

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

Figure 9-6: `msgop()` System Call Example (Sheet 6 of 7)

```
157         /*Print the received message.*/
158         for(i = 0; i<=rtrn; i++)
159             putchar(rcvbuf.mtext[i]);
160     }
161     /*Check the associated data structure.*/
162     msgctl(msqid, IPC_STAT, buf);
163     /*Print the decremented number of messages.*/
164     printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165     /*Print the process id of the last receiver.*/
166     printf("The msg_lrpid = %d\n", buf->msg_lrpid);
167     /*Print the last message receive time*/
168     printf("The msg_rtime = %d\n", buf->msg_rtime);
169 }
170 }
```

Figure 9-6: `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 *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:

## Semaphores

---

- 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 1st 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 9-1.

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 *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 9-7 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 9-7: 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 9-8 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 9-8: 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
<b>semflg</b>	=	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 *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 9-9) 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 *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**.

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

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key;        /*declare as long integer*/
13     int opperm, flags, nsems;
14     int semid, opperm_flags;

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

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

---

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

```
23  /*Set the desired flags.*/
24  printf("\nEnter corresponding number to\n");
25  printf("set the desired flags:\n");
26  printf("No flags                = 0\n");
27  printf("IPC_CREAT                  = 1\n");
28  printf("IPC_EXCL                      = 2\n");
29  printf("IPC_CREAT and IPC_EXCL        = 3\n");
30  printf("                Flags        = ");
31  /*Get the flags to be set.*/
32  scanf("%d", &flags);

33  /*Error checking (debugging)*/
34  printf ("\nkey =0x%x, opperm = 0%, flags = 0%\n",
35         key, opperm, flags);
36  /*Incorporate the control fields (flags) with
37     the operation permissions.*/
38  switch (flags)
39  {
40  case 0: /*No flags are to be set.*/
41         opperm_flags = (opperm | 0);
42         break;
43  case 1: /*Set the IPC_CREAT flag.*/
44         opperm_flags = (opperm | IPC_CREAT);
45         break;
46  case 2: /*Set the IPC_EXCL flag.*/
47         opperm_flags = (opperm | IPC_EXCL);
48         break;
49  case 3: /*Set the IPC_CREAT and IPC_EXCL
50         flags.*/
51         opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52  }
```

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

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

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

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

62      /*Perform the following if call is unsuccessful.*/
63      if(semid == -1)
64      {
65          printf("The semget system call failed!\n");
66          printf("The error number = %d\n", errno);
67      }
68      /*Return the semid upon successful completion.*/
69      else
70          printf("\nThe semid = %d\n", semid);
71      exit(0);
72 }
```

Figure 9-9: `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 *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 9-10) 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 *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`.

```
1  /*This is a program to illustrate
2  **the semaphore control, semctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     struct semid_ds semid_ds;
15     int c, i, length;
16     int uid, gid, mode;
17     int retrn, semid, semnum, cmd, choice;
18     union semun {
19         int val;
20         struct semid_ds *buf;
21         ushort array[25];
22     } arg;

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

Figure 9-10: **semctl()** System Call Example (Sheet 1 of 7)

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

28      /*Choose the desired command.*/
29      printf("\nEnter the number for\n");
30      printf("the desired cmd:\n");
31      printf("GETVAL      = 1\n");
32      printf("SETVAL      = 2\n");
33      printf("GETPID     = 3\n");
34      printf("GETNCNT    = 4\n");
35      printf("GETZCNT    = 5\n");
36      printf("GETALL     = 6\n");
37      printf("SETALL     = 7\n");
38      printf("IPC_STAT   = 8\n");
39      printf("IPC_SET    = 9\n");
40      printf("IPC_RMID   = 10\n");
41      printf("Entry     = ");
42      scanf("%d", &cmd);

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

46      /*Set the command and do the call.*/
47      switch (cmd)
48      {
```

Figure 9-10: `semctl()` System Call Example (Sheet 2 of 7)

---

```
49     case 1: /*Get a specified value.*/
50         printf("\nEnter the semnum = ");
51         scanf("%d", &semnum);
52         /*Do the system call.*/
53         retrn = semctl(semid, semnum, GETVAL, 0);
54         printf("\nThe semval = %d\n", retrn);
55         break;
56     case 2: /*Set a specified value.*/
57         printf("\nEnter the semnum = ");
58         scanf("%d", &semnum);
59         printf("\nEnter the value = ");
60         scanf("%d", &arg.val);
61         /*Do the system call.*/
62         retrn = semctl(semid, semnum, SETVAL, arg.val);
63         break;
64     case 3: /*Get the process ID.*/
65         retrn = semctl(semid, 0, GETPID, 0);
66         printf("\nThe sempid = %d\n", retrn);
67         break;
68     case 4: /*Get the number of processes
69             waiting for the semaphore to
70             become greater than its current
71             value.*/
72         printf("\nEnter the semnum = ");
73         scanf("%d", &semnum);
74         /*Do the system call.*/
75         retrn = semctl(semid, semnum, GETNCNT, 0);
76         printf("\nThe semncnt = %d", retrn);
77         break;
```

Figure 9-10: `semctl()` System Call Example (Sheet 3 of 7)

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

87     case 6: /*Get all of the semaphores.*/
88     /*Get the number of semaphores in
89     the semaphore set.*/
90     retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91     length = arg.buf->sem_nsems;
92     if(retrn == -1)
93     goto ERROR;
94     /*Get and print all semaphores in the
95     specified set.*/
96     retrn = semctl(semid, 0, GETALL, arg.array);
97     for (i = 0; i < length; i++)
98     {
99         printf("%d", arg.array[i]);
100        /*Seperate each
101        semaphore.*/
102        printf("%c", ' ');
103    }
104    break;
```

Figure 9-10: semctl() System Call Example (Sheet 4 of 7)

---



```
105     case 7: /*Set all semaphores in the set.*/
106           /*Get the number of semaphores in
107             the set.*/
108           retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109           length = arg.buf->sem_nsems;
110           printf("Length = %d\n", length);
111           if(retrn == -1)
112             goto ERROR;
113           /*Set the semaphore set values.*/
114           printf("\nEnter each value:\n");
115           for(i = 0; i < length ; i++)
116             {
117               scanf("%d", &c);
118               arg.array[i] = c;
119             }
120           /*Do the system call.*/
121           retrn = semctl(semid, 0, SETALL, arg.array);
122           break;

123     case 8: /*Get the status for the semaphore set.*/
124           /*Get and print the current status values.*/
125           retrn = semctl(semid, 0, IPC_STAT, arg.buf);
126           printf ("\nThe USER ID = %d\n",
127                 arg.buf->sem_perm.uid);
128           printf ("The GROUP ID = %d\n",
129                 arg.buf->sem_perm.gid);
130           printf ("The operation permissions = 0%o\n",
131                 arg.buf->sem_perm.mode);
132           printf ("The number of semaphores in
133                 set = %d\n", arg.buf->sem_nsems);
134           printf ("The last semop time = %d\n",
135                 arg.buf->sem_otime);
136
137
```

Figure 9-10: `semctl()` System Call Example (Sheet 5 of 7)

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

141     case 9: /*Select and change the desired
142             member of the data structure.*/
143             /*Get the current status values.*/
144             retn = semctl(semid, 0, IPC_STAT, arg.buf);
145             if(retn == -1)
146                 goto ERROR;
147             /*Select the member to change.*/
148             printf("\nEnter the number for the\n");
149             printf("member to be changed:\n");
150             printf("sem_perm.uid   = 1\n");
151             printf("sem_perm.gid   = 2\n");
152             printf("sem_perm.mode  = 3\n");
153             printf("Entry         = ");
154             scanf("%d", &choice);
155             switch(choice){

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

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

Figure 9-10: `semctl()` System Call Example (Sheet 6 of 7)

```
170         case 3: /*Change the mode portion of
171                 the operation
172                     permissions.*/
173                 printf("\nEnter MODE = ");
174                 scanf("%o", &mode);
175                 arg.buf->sem_perm.mode = mode;
176                 printf("\nMODE = 0%o\n",
177                         arg.buf->sem_perm.mode);
178                 break;
179             }
180             /*Do the change.*/
181             retrn = semctl(semid, 0, IPC_SET, arg.buf);
182             break;
183         case 10: /*Remove the semid along with its
184                 data structure.*/
185             retrn = semctl(semid, 0, IPC_RMID, 0);
186             }
187             /*Perform following if call unsuccessful.*/
188             if(retrn == -1)
189             {
190             ERROR:
191                 printf ("\n\nsemctl system call failed!\n");
192                 printf ("The error number = %d\n", errno);
193                 exit(0);
194             }
195             printf ("\n\nsemctl system call successful\n");
196             printf ("for semid = %d\n", semid);
197             exit (0);
198     }
```

Figure 9-10: `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 *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. **Sembuf** is the tag name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file.

The **nsops** argument specifies the length of the array (the number of structures in the array). The maximum size of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each **semop(2)** system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value
- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- **IPC\_NOWAIT** – this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which **IPC\_NOWAIT** is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.

- **SEM\_UNDO** – this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the **IPC\_NOWAIT** flag set. That is, the blocked operation waits until it can perform its operation; and when it and all succeeding operations are successful, all operations with the **SEM\_UNDO** flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

### Example Program

The example program in this section (Figure 9-11) is a menu driven program which allows all possible combinations of using the **semop(2)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop(2)** entry in the *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). **Sembuf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl()** 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**.



```
1  /*This is a program to illustrate
2  **the semaphore operations, semop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     struct sembuf sembuf[10], *sops;
15     char string[];
16     int retrn, flags, sem_num, i, semid;
17     unsigned nsops;
18     sops = sembuf; /*Pointer to array sembuf.*/

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

Figure 9-11: **semop(2)** System Call Example (Sheet 1 of 4)

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

30      /*Initialize the array for the
31         number of operations to be performed.*/
32      for(i = 0; i < nsops; i++, sops++)
33      {

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

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

Figure 9-11: **semop(2)** System Call Example (Sheet 2 of 4)

```
51      /*Specify the desired flags.*/
52      printf("\nEnter the corresponding\n");
53      printf("number for the desired\n");
54      printf("flags:\n");
55      printf("No flags           = 0\n");
56      printf("IPC_NOWAIT          = 1\n");
57      printf("SEM_UNDO            = 2\n");
58      printf("IPC_NOWAIT and SEM_UNDO = 3\n");
59      printf("           Flags      = ");
60      scanf("%d", &flags);

61      switch(flags)
62      {
63      case 0:
64          sops->sem_flg = 0;
65          break;
66      case 1:
67          sops->sem_flg = IPC_NOWAIT;
68          break;
69      case 2:
70          sops->sem_flg = SEM_UNDO;
71          break;
72      case 3:
73          sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74          break;
75      }
76      printf("\nFlags = 0%o\n", sops->sem_flg);
77  }
```

Figure 9-11: **semop(2)** System Call Example (Sheet 3 of 4)

```
78     /*Print out each structure in the array.*/
79     for(i = 0; i < nsops; i++)
80     {
81         printf("\nsem_num = %d\n", sembuf[i].sem_num);
82         printf("sem_op = %d\n", sembuf[i].sem_op);
83         printf("sem_flg = %o\n", sembuf[i].sem_flg);
84         printf("%c", ' ');
85     }

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

88     /*Do the semop system call.*/
89     retrn = semop(semid, sops, nsops);
90     if(retrn == -1) {
91         printf("\nSemop failed.  ");
92         printf("Error = %d\n", errno);
93     }
94     else {
95         printf ("\nSemop was successful\n");
96         printf("for semid = %d\n", semid);

97         printf("Value returned = %d\n", retrn);
98     }
99 }
```

Figure 9-11: **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 *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 9-1.

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 9-12 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 9-12: Shared Memory State Information

---

The implied states of Figure 9-12 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 *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

## Shared Memory

---

- **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 9-13 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 9-13: 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:

<b>SHM_R</b>	0400
<b>SHM_W</b>	0200

Control commands are predefined constants (represented by all uppercase letters). Figure 9-14 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 9-14: 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:

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

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

As specified by the **shmget(2)** entry in the *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:

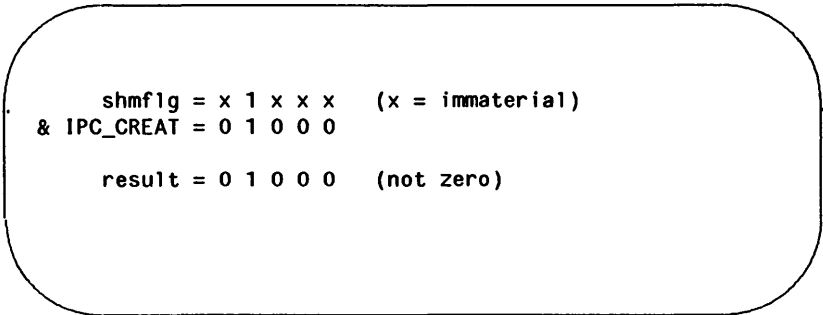
```
shmid = shmget (IPC_PRIVATE, size, shmflg);
```

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 (**shmid**s) in the UNIX operating system.

The second condition is satisfied if the value for **key** is not already associated with a **shmid** and the bitwise ANDing of **shmflg** and IPC\_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC\_CREAT flag is set (**shmflg** ; IPC\_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:



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 9-15) 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 *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.

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

4  #include <sys/types.h>
5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <errno.h>

8  /*Start of main C language program*/
9  main()
10 {
11     key_t key;                /*declare as long integer*/
12     int opperm, flags;
13     int shmid, size, opperm_flags;
14     /*Enter the desired key*/
15     printf("Enter the desired key in hex = ");
16     scanf("%x", &key);

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

Figure 9-15: `shmget(2)` System Call Example (Sheet 1 of 3)

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

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

35  /*Incorporate the control fields (flags) with
36  the operation permissions*/
37  switch (flags)
38  {
39  case 0:  /*No flags are to be set.*/
40          opperm_flags = (opperm | 0);
41          break;
42  case 1:  /*Set the IPC_CREAT flag.*/
43          opperm_flags = (opperm | IPC_CREAT);
44          break;
45  case 2:  /*Set the IPC_EXCL flag.*/
46          opperm_flags = (opperm | IPC_EXCL);
47          break;
48  case 3:  /*Set the IPC_CREAT and IPC_EXCL flags.*/
49          opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50  }
```

Figure 9-15: shmget(2) System Call Example (Sheet 2 of 3)

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

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

57  /*Perform the following if the call is unsuccessful.*/
58  if(shmids == -1)
59  {
60      printf ("\nThe shmget system call failed!\n");
61      printf ("The error number = %d\n", errno);
62  }
63  /*Return the shmids upon successful completion.*/
64  else
65      printf ("\nThe shmids = %d\n", shmids);
66  exit(0);
67  }
```

Figure 9-15: `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 *Programmer's Reference Manual* is as follows:

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

int shmctl (shmids, cmd, buf)
int shmids, cmd;
struct shmids *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 **shmids** 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 **shmids** and place it in the data structure pointed to by the **\*buf** pointer in the user memory area
- **IPC\_SET** – for the specified **shmids**, set the effective user and group identification, and operation permissions
- **IPC\_RMID** – remove the specified **shmids** 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 9-16) 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 *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.



```
1  /*This is a program to illustrate
2  **the shared memory control, shmctl(),
3  **system call capabilities.
4  */
5
6  /*Include necessary header files.*/
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/ipc.h>
10 #include <sys/shm.h>
11
12 /*Start of main C language program*/
13 main()
14 {
15     extern int errno;
16     int uid, gid, mode;
17     int rtn, shmid, command, choice;
18     struct shmctl_ds shmctl_ds, *buf;
19     buf = &shmctl_ds;
20
21     /*Get the shmid, and command.*/
22     printf("Enter the shmid = ");
23     scanf("%d", &shmid);
24     printf("\nEnter the number for\n");
25     printf("the desired command:\n");
```

Figure 9-16: **shmctl(2)** System Call Example (Sheet 1 of 6)

```
23     printf("IPC_STAT   = 1\n");
24     printf("IPC_SET    = 2\n");
25     printf("IPC_RMID   = 3\n");
26     printf("SHM_LOCK   = 4\n");
27     printf("SHM_UNLOCK = 5\n");
28     printf("Entry     = ");
29     scanf("%d", &command);

30     /*Check the values.*/
31     printf ("\nshmid =%d, command = %d\n",
32           shmids, command);

33     switch (command)
34     {
35     case 1: /*Use shmctl() to duplicate
36            the data structure for
37            shmids in the shmids area pointed
38            to by buf and then print it out.*/
39         rtrn = shmctl(shmids, IPC_STAT,
40                     buf);
41         printf ("\nThe USER ID = %d\n",
42               buf->shm_perm.uid);
43         printf ("The GROUP ID = %d\n",
44               buf->shm_perm.gid);
45         printf ("The creator's ID = %d\n",
46               buf->shm_perm.cuid);
47         printf ("The creator's group ID = %d\n",
48               buf->shm_perm.cgid);
49         printf ("The operation permissions = 0%o\n",
50               buf->shm_perm.mode);
51         printf ("The slot usage sequence\n");
```

Figure 9-16: 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 9-16: **shmctl(2)** System Call Example (Sheet 3 of 6)

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

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

93                printf("\nEnter the number for the\n");
94                printf("member to be changed:\n");
95                printf("shm_perm.uid   = 1\n");
96                printf("shm_perm.gid   = 2\n");
97                printf("shm_perm.mode = 3\n");
98                printf("Entry       = ");
99                scanf("%d", &choice);
100                /*Only one choice is allowed per
101                pass as an illegal entry will
102                cause repetitive failures until
103                shmId_ds is updated with
104                IPC_STAT.*/
```

Figure 9-16: shmctl(2) System Call Example (Sheet 4 of 6)

```
105         switch(choice){
106         case 1:
107             printf("\nEnter USER ID = ");
108             scanf ("%d", &uid);
109             buf->shm_perm.uid = uid;
110             printf("\nUSER ID = %d\n",
111                 buf->shm_perm.uid);
112             break;

113         case 2:
114             printf("\nEnter GROUP ID = ");
115             scanf ("%d", &gid);
116             buf->shm_perm.gid = gid;
117             printf("\nGROUP ID = %d\n",
118                 buf->shm_perm.gid);
119             break;

120         case 3:
121             printf("\nEnter MODE = ");
122             scanf ("%o", &mode);
123             buf->shm_perm.mode = mode;
124             printf("\nMODE = 0%o\n",
125                 buf->shm_perm.mode);
126             break;
127         }
128         /*Do the change.*/
129         rtrn = shmctl(shmid, IPC_SET,
130                     buf);
131         break;
```

Figure 9-16: `shmctl()` System Call Example (Sheet 5 of 6)

```
132     case 3: /*Remove the shmid along with its
133             associated
134             data structure.*/
135         rtn = shmctl(shmid, IPC_RMID, NULL);
136         break;

137     case 4: /*Lock the shared memory segment*/
138         rtn = shmctl(shmid, SHM_LOCK, NULL);
139         break;
140     case 5: /*Unlock the shared memory
141             segment.*/
142         rtn = shmctl(shmid, SHM_UNLOCK, NULL);
143         break;
144     }
145     /*Perform the following if call is unsuccessful.*/
146     if(rtn == -1)
147     {
148         printf ("\nThe shmctl system call failed!\n");
149         printf ("The error number = %d\n", errno);
150     }
151     /*Return the shmid upon successful completion.*/
152     else
153         printf ("\nShmctl was successful for
154             shmid = %d\n", shmid);
155     exit (0);
156 }
```

---

Figure 9-16: **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 *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 **shmaddr** argument can be zero or user supplied when passed to the **shmat(2)** system call. If it is zero, the UNIX operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the UNIX operating system would pick. The following illustrates some typical address ranges; these are for the 3B2 Computer:

```
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 9-17) 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 *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.

```
1  /*This is a program to illustrate
2  **the shared memory operations, shmop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int flags, addr, i, attach;
15     int shmid, shmflg, retrn, detach;

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

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

Figure 9-17: **shmop()** System Call Example (Sheet 1 of 4)

---

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

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

37         /*Specify the desired flags.*/
38         printf("\nEnter the corresponding\n");
39         printf("number for the desired\n");
40         printf("flags:\n");
41         printf("SHM_RND                = 1\n");
42         printf("SHM_RDONLY                = 2\n");
43         printf("SHM_RND and SHM_RDONLY = 3\n");
44         printf("          Flags          = ");
45         scanf("%d", &flags);
```

Figure 9-17: `shmop()` System Call Example (Sheet 2 of 4)

```
46         switch(flags)
47         {
48         case 1:
49             shmflg = SHM_RND;
50             break;
51         case 2:
52             shmflg = SHM_RDONLY;
53             break;
54         case 3:
55             shmflg = SHM_RND | SHM_RDONLY;
56             break;
57         }
58         printf("\nFlags = 0%o\n", shmflg);

59         /*Do the shmat system call.*/
60         retrn = (int)shmat(shmid, addr, shmflg);
61         if(retrn == -1) {
62             printf("\nShmat failed. ");
63             printf("Error = %d\n", errno);
64         }
65         else {
66             printf ("\nShmat was successful\n");
67             printf("for shmid = %d\n", shmid);
68             printf("The address = 0x%x\n", retrn);
69         }
70     }

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

Figure 9-17: `shmop()` System Call Example (Sheet 3 of 4)

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

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

96         /*Do the shmdt system call.*/
97         retrn = (int)shmdt(addr);
98         if(retrn == -1) {
99             printf("Error = %d\n", errno);
100        }
101        else {
102            printf ("\nShmdt was successful\n");
103            printf("for address = 0%x\n", addr);

104        }

105    }
106 }
```

Figure 9-17: `shmop()` System Call Example (Sheet 4 of 4)





---

# Chapter 10: curses/terminfo

Introduction	10-1
Overview	10-3
What is <b>curses</b> ?	10-3
What is <b>terminfo</b> ?	10-5
How <b>curses</b> and <b>terminfo</b> Work Together	10-6
Other Components of the Terminal Information Utilities	10-7
Working with <b>curses</b> Routines	10-9
What Every <b>curses</b> Program Needs	10-9
The Header File <b>&lt;curses.h&gt;</b>	10-9
The Routines <b>initscr()</b> , <b>refresh()</b> , <b>endwin()</b>	10-11
Compiling a <b>curses</b> Program	10-12
Running a <b>curses</b> Program	10-13
More about <b>initscr()</b> and Lines and Columns	10-14
More about <b>refresh()</b> and Windows	10-14
Getting Simple Output and Input	10-19
Output	10-19
Input	10-31
Controlling Output and Input	10-39
Output Attributes	10-39
Bells, Whistles, and Flashing Lights	10-44
Input Options	10-45
Building Windows and Pads	10-50
Output and Input	10-50
The Routines <b>wnoutrefresh()</b> and <b>doupdate()</b>	10-51
New Windows	10-56

## Table of Contents

---

Using Advanced <b>curses</b> Features	10-59
Routines for Drawing Lines and Other Graphics	10-59
Routines for Using Soft Labels	10-61
Working with More than One Terminal	10-62
Working with <b>terminfo</b> Routines	10-65
What Every <b>terminfo</b> Program Needs	10-65
Compiling and Running a <b>terminfo</b> Program	10-67
An Example <b>terminfo</b> Program	10-67
Working with the <b>terminfo</b> Database	10-71
Writing Terminal Descriptions	10-71
Name the Terminal	10-72
Learn About the Capabilities	10-72
Specify Capabilities	10-73
Compile the Description	10-80
Test the Description	10-81
Comparing or Printing <b>terminfo</b> Descriptions	10-82
Converting <b>termcap</b> Description to a <b>terminfo</b> Description	10-82
<b>curses</b> Program Examples	10-84
The <b>editor</b> Program	10-84
The <b>highlight</b> Program	10-91
The <b>scatter</b> Program	10-93
The <b>show</b> Program	10-96
The <b>two</b> Program	10-98
The <b>window</b> Program	10-101

---

# Introduction

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the Terminal Information Utilities package, commonly called **curses/terminfo**, to write screen management programs on a UNIX system. This package includes a library of C routines, a database, and a set of UNIX system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to **curses(3X)** and **terminfo(4)** in the *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.

---

# Overview

## What is curses?

**curses(3X)** is the library of routines that you use to write screen management programs on the UNIX system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw()** that behaves much like **printf(3S)** and another routine **getch()** that behaves like **getc(3S)**. The automatic teller program at your bank might use **printw()** to print its menus and **getch()** to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX system screen editor **vi(1)** might also use these and other **curses** routines.

The **curses** routines are usually located in **/usr/lib/libcurses.a**. 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 10-1: A Simple **curses** Program

---

## What Is terminfo?

**terminfo** refers to both of the following:

- It is a group of routines within the **curses** library that handles certain terminal capabilities. You can use these routines to program function keys, if your terminal has programmable keys, or write filters, for example. Shell programmers, as well as C programmers, can use the **terminfo** routines in their programs.
- It is a database containing the descriptions of many terminals that can be used with **curses** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations – for example, how many lines and columns it has and how its control characters are interpreted.

Each terminal description in the database is a separate, compiled file. You use the source code that **terminfo(4)** describes to create these files and the command **tic(1M)** to compile them.

The compiled files are normally located in the directories **/usr/lib/terminfo/?**. These directories have single character names, each of which is the first character in the name of a terminal. For example, an entry for the AT&T Teletype 5425 is normally located in the file **/usr/lib/terminfo/a/att5425**.

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 10-2: 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 AT&T Teletype 5425 terminal to run the simple **curses** program shown in Figure 10-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the Bu11sEye in the middle of it. The description of the AT&T Teletype 5425 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 *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:

<b>captainfo(1M)</b>	a tool for converting terminal descriptions developed on earlier releases of the UNIX system to <b>terminfo</b> descriptions
<b>curses(3X)</b>	
<b>infocmp(1M)</b>	a tool for printing and comparing compiled terminal descriptions
<b>tabs(1)</b>	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 *Programmer's Reference Manual* and the *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()`, and `endwin()`**

The routines `initscr()`, `refresh()`, and `endwin()` initialize a terminal screen to an "in **curses** state," update the contents of the screen, and restore the terminal to an "out of **curses** state," respectively. Use the simple program that we introduced earlier to learn about each of these routines:

```
#include <curses.h>

main()
{
    initscr(); /*initialize terminal settings & <curses.h>
               data structures and variables */

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh(); /* send output to (update) screen */
    addstr("Eye");
    refresh(); /* send more output to terminal screen */
    endwin(); /* restore all terminal settings */
}
```

Figure 10-3: 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("Bu11s");
```

says to write the character string **Bu11s**. 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 *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 **sdb(1)**, which is documented in the *Programmer's Reference Manual*). When using **sdb**, you have to keep a few points in mind. First, a **curses** program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **sdb**, however, may cause changes to the contents of the screen of which the **curses** program is not aware.

Second, a **curses** program outputs to a window until **refresh()** or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on **curses** routines that are macros, such as **refresh()**, does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh()** instead of **refresh()**. See the above section, "The Header File `<curses.h>`," for more information about macros.

### More about **initscr()** and Lines and Columns

After determining a terminal's screen dimensions, **initscr()** sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment **\$LINES** and **\$COLUMNS**.

### More about **refresh()** and Windows

As mentioned above, **curses** routines do not update a terminal until **refresh()** is called. Instead, they write to an internal representation of the screen called a window. When **refresh()** is called, all the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like a buffer does when you use a UNIX system editor. When you invoke **vi(1)**, for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of **curses** routines, they change the



contents of a window. The changes become part of the current terminal screen only when **refresh()** is called.

< **curses.h** > supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using **curses** routines. The header file defines **stdscr** to be of the type **WINDOW\***, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh()** is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like **stdscr**. A **curses** program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as is possible. Figure 10-4 illustrates what happens when you execute the sample **curses** program that prints **Bullseye** at the center of a terminal screen (see Figure 10-1). Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh()** is called. This **refresh()** clears the screen and updates it with the current contents of **stdscr**.

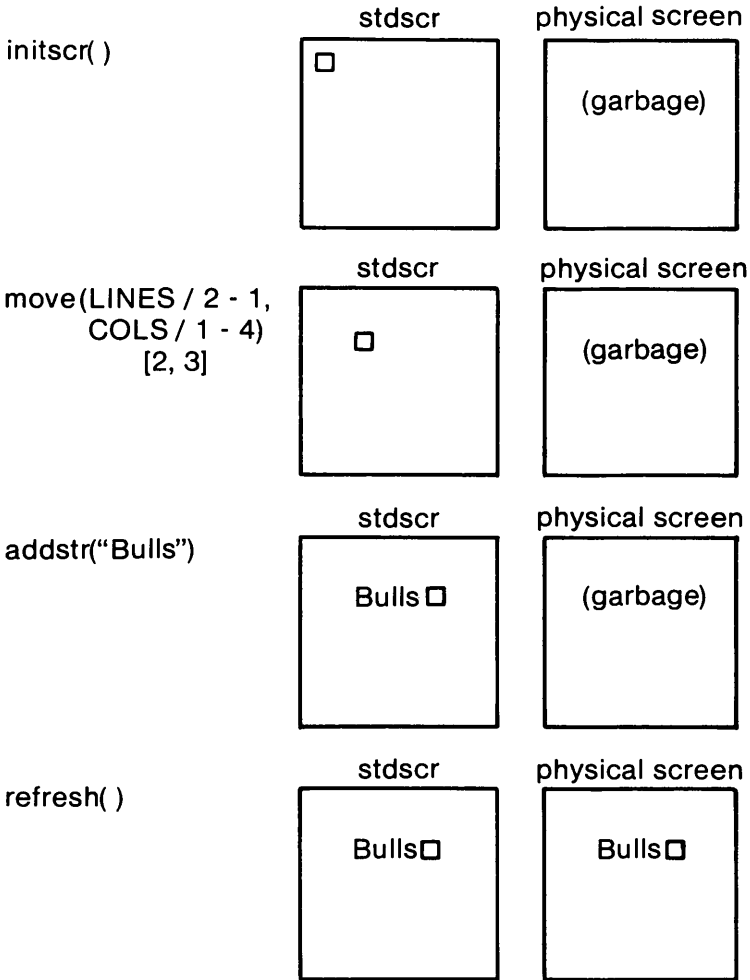


Figure 10-4: The Relationship between **stdscr** and a Terminal Screen (Sheet 1 of 2)

---

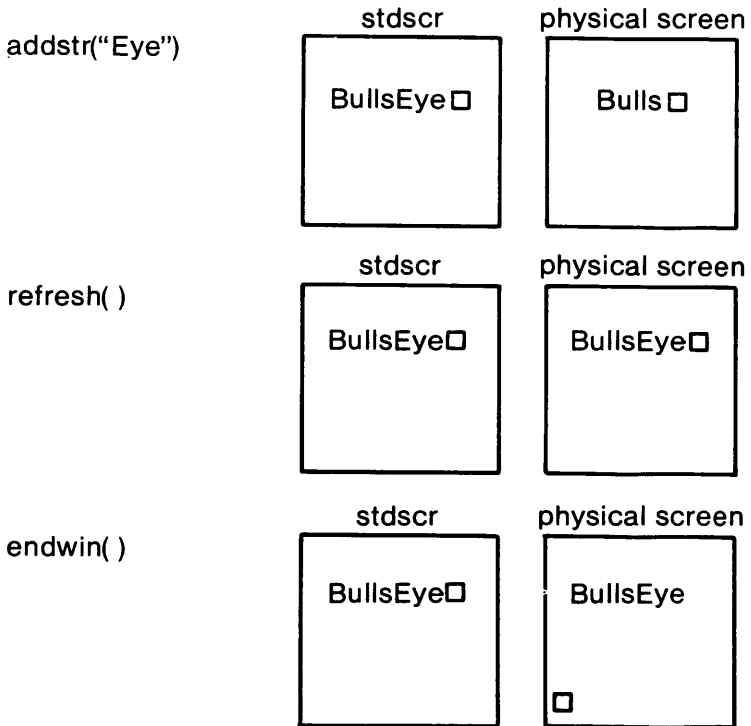


Figure 10-4: The Relationship Between `stdscr` and a Terminal Screen (Sheet 2 of 2)

---

You can create other windows and use them instead of `stdscr`. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It's possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It's also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some `curses` routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 10-5 represents what a pad, a subwindow, and some other windows could look like in comparison to a terminal screen.

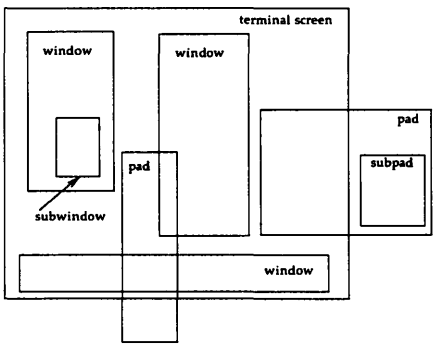


Figure 10-5: Multiple Windows and Pads Mapped to a Terminal Screen

---

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()**, **clrtoobot()**

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

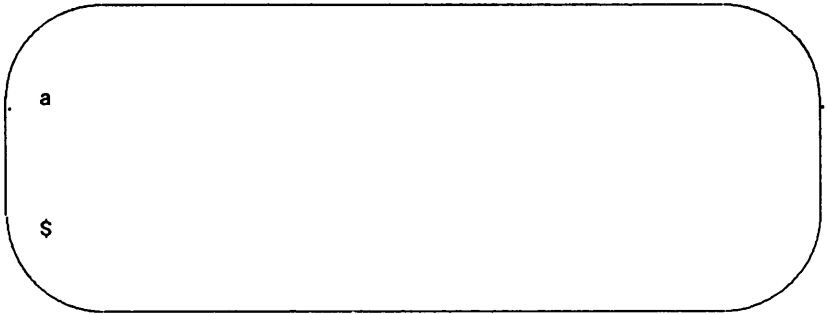
- **addch()** writes a single character to **stdscr**.
- The character is of the type **chtype**, which is defined in **< curses.h >**. **chtype** contains data and attributes (see "Output Attributes" in this chapter for information about attributes).
- When working with variables of this type, make 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:



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 **Bu11sEye**. See Figures 10-1, 10-2, and 10-4.



## **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**

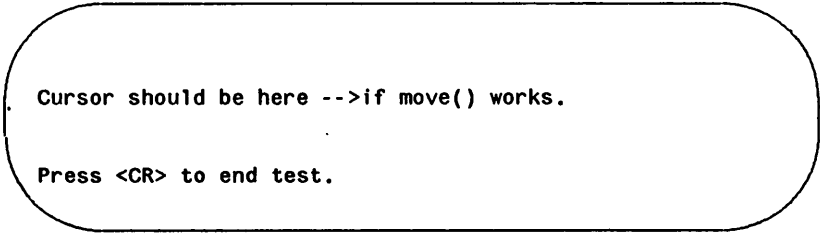
- **move()** positions the cursor for **stdscr** at the given row **y** and the given column **x**.
- Notice that **move()** takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section "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>

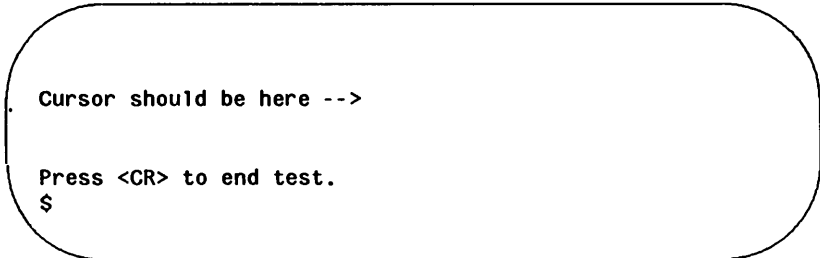
main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();    /* Gets <CR>; discussed below. */
    endwin();
}
```

Here's the output generated by running this program:



Cursor should be here -->if move() works.  
Press <CR> to end test.

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



Cursor should be here -->  
Press <CR> to end test.  
\$

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

**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 `clrtoebot()`.

```
#include < curses.h>

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

Here's the output generated by running this program:



```
Press <CR> to delete from hereto the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to `refresh()`: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press `<CR>`:

Press <CR> to delete from here

\$

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



## **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)**.

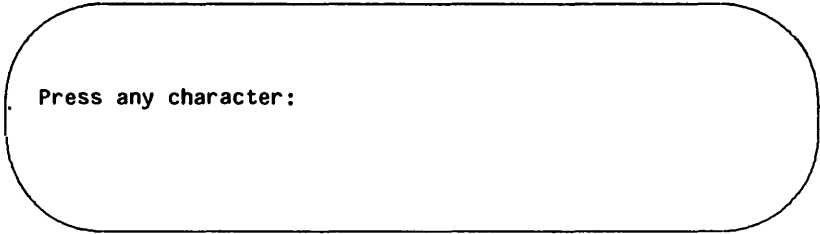
## EXAMPLE

```
#include < curses.h>

main()
{
    int ch;

    initscr();
    cbreak(); /* Explained in the section "Input Options" */
    addstr("Press any character: ");
    refresh();
    ch = getch();
    printw("\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 **<CR>**, **<NL>**, or **<ENTER>** is received from **stdscr**. **getstr()** does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr()** is a macro; it calls **getch()** to read each character.
- **getstr()** returns **ERR** if **getch()** returns **ERR** to it. Otherwise it returns **OK**.
- See the discussions about **echo()**, **noecho()**, **cbreak()**, **nocbreak()**, **raw()**, **noraw()**, **halfdelay()**, **nodelay()**, and **keypad()** below and in **curses(3X)**.

### EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

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

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

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

```
I enjoy learning about the UNIX system.
```

```
The string entered was
```

```
'I enjoy learning about the UNIX system.'
```

```
$
```

## **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 number and string separated by a comma: ");
    refresh();
    scanw("%f,%s",&number,string);
    clear();
   printw("String was \"%s\" and 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 off all attributes.

In addition to the attributes listed above, there are two bit masks called `A_CHARTEXT` and `A_ATTRIBUTES`. You can use these bit masks with the **curses** function `inch()` and the C logical AND ( `&` ) operator to extract the character or attributes of a position on a terminal screen. See the discussion of `inch()` on the **curses(3X)** manual page.

Following are descriptions of **attrset()** and the other **curses** routines that you can use to manipulate attributes.

**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 terminal's screen, if possible, and, if not, tries to ring the terminal bell.
- **beep()** tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- Neither returns any useful value.

## Input Options

The UNIX system does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed
- interprets 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 `<CR>` to `<NL>`

Because a **curses** program maintains total control over the screen, **curses** turns off echoing on the UNIX system and does echoing itself. At times, you may not want the UNIX system to process other characters in the standard way in an interactive screen management program. Some **curses** routines, **noecho()** and **cbreak()**, for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Figure 10-6 shows some of the major routines for controlling input.

Every **curses** program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak()**, **raw()**, **nocbreak()**, or **noraw()** mode. Although the **curses** program starts up in **echo()** mode, as Figure 10-6 shows, none of the other modes are guaranteed.

The combination of **noecho()** and **cbreak()** is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The **curses** routine **noecho()** is designed for this purpose. However, when **noecho()** turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak()** causes these characters to

be uninterpreted.

Input Options	Characters	
	Interpreted	Uninterpreted
Normal 'out of curses state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal curses 'start up state'	echoing (simulated)	All else undefined.
<b>cbreak()</b> and <b>echo()</b>	interrupt, quit stripping echoing	erase, kill EOF
<b>cbreak()</b> and <b>noecho()</b>	interrupt, quit stripping	echoing erase, kill EOF
<b>nocbreak()</b> and <b>noecho()</b>	break, quit stripping erase, kill EOF	echoing
<b>nocbreak()</b> and <b>echo()</b>	See caution below.	
<b>nl()</b>	<CR> to <NL>	
<b>nonl()</b>		<CR> to <NL>
<b>raw()</b> (instead of <b>cbreak()</b> )		break, quit stripping

Figure 10-6: 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 10-6, you can use the **curses** routines **noraw()**, **halfdelay()**, and **nodelay()** to control input. See the **curses(3X)** manual page for discussions of these routines.

The next few pages describe **noecho()**, **cbreak()** and the related routines **echo()** and **nocbreak()** in more detail.

### **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 10-6 and accompanying caution. After you turn echoing off, you can still echo characters with **addch()**.

#### EXAMPLE

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

**`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 10-6 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...])`
- `mvprintw(win, y, x, fmt [, arg...])`
- `wmove(win, y, x)`
- `wclear(win)` and `werase(win)`
- `wclrtoeol(win)` and `wclrbot(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.

Figure 10-7 illustrates the effect of **wnoutrefresh()** and **doupdate()** on these two windows, the virtual screen, and the physical screen:

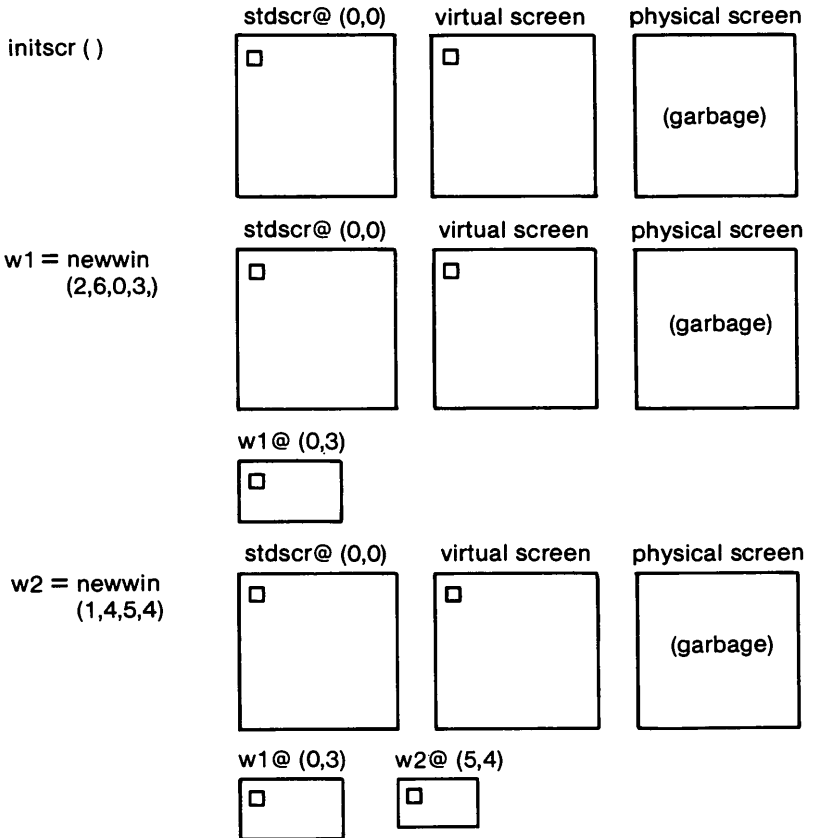


Figure 10-7: The Relationship Between a Window and a Terminal Screen (Sheet 1 of 3)

---

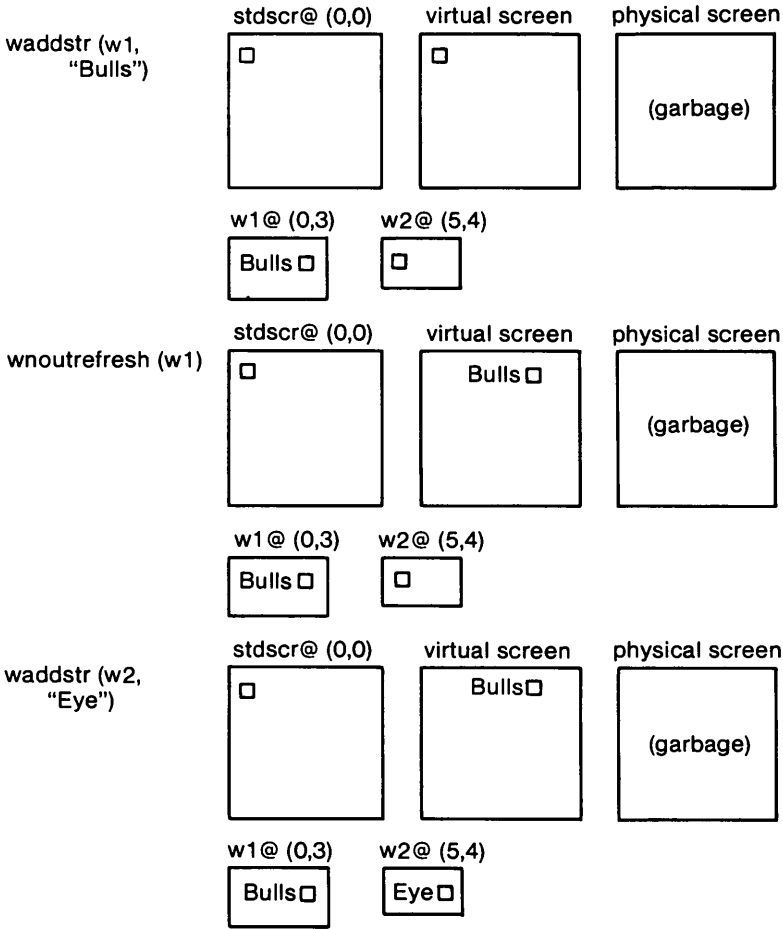


Figure 10-7: The Relationship Between a Window and a Terminal Screen (Sheet 2 of 3)

---



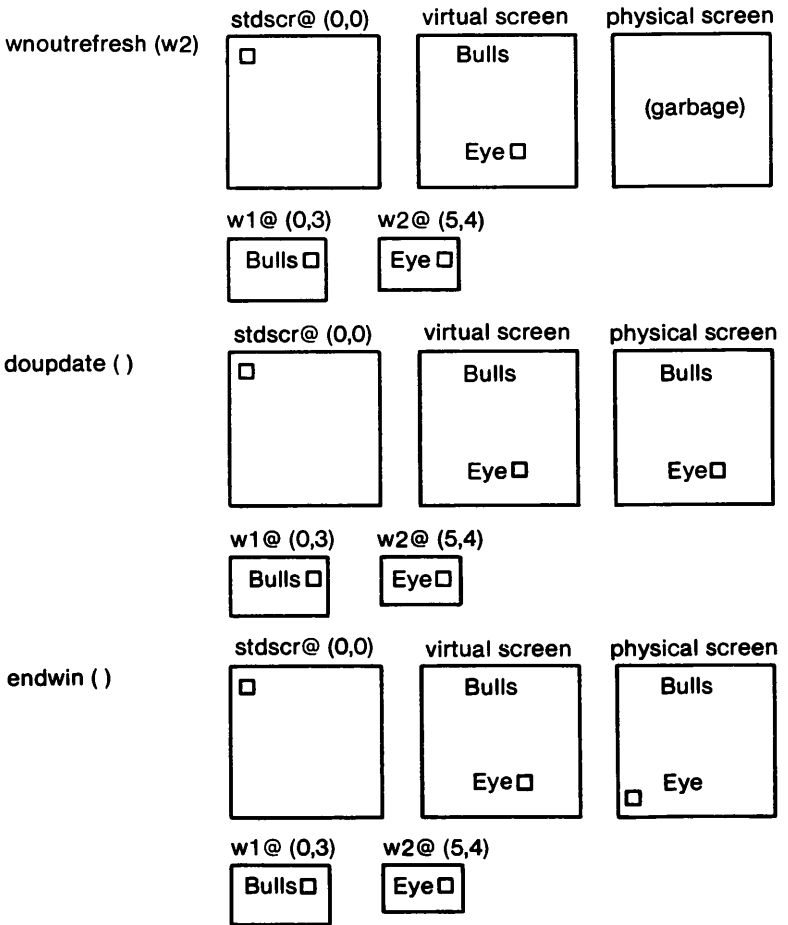


Figure 10-7: The Relationship Between a Window and a Terminal Screen (Sheet 3 of 3)

### New Windows

Following are descriptions of the routines `newwin()` and `subwin()`, which you use to create new windows. For information about creating new pads with `newpad()` and `subpad()`, see the `curses(3X)` manual page.

#### `newwin()`

#### SYNOPSIS

```
#include < curses.h >
```

```
WINDOW *newwin(nlines, ncols, begin_y, begin_x)  
int nlines, ncols, begin_y, begin_x;
```

#### NOTES

- `newwin()` returns a pointer to a new window with a new data area.
- The variables `nlines` and `ncols` give the size of the new window.
- `begin_y` and `begin_x` give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

#### EXAMPLE

Recall the sample program using two windows; see Figure 10-7. Also see the `window` program under "curses Program Examples" in this chapter.

## **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 *Programmer's Guide*).

### EXAMPLE

```
#include < curses.h>

main()
{
    WINDOW *sub;

    initscr();
    box(stdscr, 'w', 'w'); /*See box() on curses(3X) manual pg.*/
    mvwaddstr(stdscr, 7, 10, "----- this is 10,10");
    mvwaddch(stdscr, 8, 10, '|');
    mvwaddch(stdscr, 9, 10, 'v');
    sub = subwin(stdscr, 10, 20, 10, 10);
    box(sub, 's', 's');
    wnoutrefresh(stdscr);
    wrefresh(sub);
    endwin();
}
```

This program prints a border of **w**s around the **stdscr** (the sides of your terminal screen) and a border of **s**'s around the subwindow **sub** when it is run. For another example, see the **win-  
dow** 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 *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 10-8 shows a typical scenario to output a message to several terminals.

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 10-8: Sending a Message to Several Terminals

---

See the **two** program under "**curses** Program Examples" in this chapter for a more complete example.

---

# Working with `terminfo` Routines

Some programs need to use lower level routines (i.e., primitives) than those offered by the `curses` routines. For such programs, the `terminfo` routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use `terminfo` routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the `terminfo` routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines: the higher level `curses` routines make your program more portable to other UNIX systems and to a wider class of terminals.

## **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 10-9.

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

Figure 10-9: Typical Framework of a **terminfo** Program

---

The header files **<curses.h>** and **<term.h>** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm()** takes care of initialization. Passing this routine the values **(char\*)0**, **1**, and **(int\*)0** invokes reasonable defaults. If **setupterm()** can't figure out what kind of terminal you are on, it prints an error message and exits. **reset\_shell\_mode()** performs functions similar to **endwin()** and should be called before a **terminfo** program exits.

A global variable like **clear\_screen** is defined by the call to **setupterm()**. It can be output using the **terminfo** routines **putp()** or **tputs()**, which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf(3S)**, because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch()** and **getch()** are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo(4)**; see **curses(3X)** for a list of all the **terminfo** routines.

## Compiling and Running a terminfo Program

The general command line for compiling and the guidelines for running a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections "Compiling a **curses** Program" and "Running a **curses** Program" in this chapter for more information.

### An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see "**curses** Program Examples") that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * 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)
    {
```

- CONTINUED -

```
fprintf(stderr, "Usage: termh1 [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;
        }
    }
}
```

```

                                - CONTINUED -
                                continue;
                                case 'N':
                                tputs(exit_attribute_mode, 1, outch);
                                ulmode = 0;
                                continue;
                                }
                                putchar(c);
                                putchar(c2);
                                }
                                else
                                putchar(c);
                                }
                                fclose(fd);
                                fflush(stdout);
                                resetterm();
                                exit(0);
                                }
                                /*
                                * Function is like putchar, but checks for underlining.
                                */
                                putchar(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);
                                }
                                }

```

**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 **tput(cap)** is a convenient abbreviation. **termhl** could be simplified by using **tput()**.

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**. 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 AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a -w) version of our fictitious terminal would be described as **myterm-w**. **term(5)** describes mode indicators in greater detail.

### Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.
- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways – type:

**stty -echo; cat -vu**

*Type in the keys you want to test;*

*for example, see what right arrow ( ) transmits.*

**<CR>**

**<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 <CR> is a CTRL-M (**cr**).

- A cursor up one line motion is a CTRL-K (**cuu1**).
- A cursor down one line motion is a CTRL-J (**cul1**).
- 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, cul1=^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:

```
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, e1=\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 **tparam()** 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:

<b>cup =</b>	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,  
sms0=\ED, rms0=\EZ, e1=\EK$<3>, ind=0  
kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,  
kcu1=\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 <CR>
```

(The trace information appears as the compilation proceeds.)

Refer to the **tic(1M)** manual page in the *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 [-Ttype] capname
```

The type of terminal you are requesting information about is identified with the **-Ttype** option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **\$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

```
tput clear
```

(The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols
```

(The number of columns used by the terminal appears here.)

The **tput(1)** manual page found in the *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**

## Examples

---

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);
}
```



```
                - CONTINUED -
/* 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;
}
```

- CONTINUED -

```
/* 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;
        }
    }
}
```

```
        - CONTINUED -
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 new line and enter input mode */
case KEY_IL:
case 'o':
    move(++row, col = 0);
    insertln();
    input();
    break;
```

- CONTINUED -

```
/* 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;
```

- CONTINUED -

```
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)
```

```

- CONTINUED -
{
    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 & 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++;
        }
    }
}
```



```
                                - CONTINUED -
                                }
                                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)
    {
```

- CONTINUED -

```
        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))
            {
                clrtoeol();
                done();
            }
            move(line, 0);
            printw("%s", linebuf);
        }
        refresh();
        if (getch() == 'q')
            done();
    }
}

void done()
{
    move(LINES - 1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

## The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

**two** is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "**sleep 100000**" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
    void done(), exit();
    unsigned sleep();
    char *getenv();
    int c;
```

- CONTINUED -

```
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 */
```

```
                - CONTINUED -
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)
            {
                clrtoeol();
                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 */
}
```

- CONTINUED -

```
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, "Line %d of stdscr", i);

    for (;;)
    {
        refresh();
        c = getch();
        switch (c)
        {
            case 'c': /*Enter command on 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);
            }
        }
    }
}
```



```
        - CONTINUED -  
        /*  
        * The command is now in buf.  
        * It should be processed here.  
        */  
  
        case 'q':  
            endwin();  
            exit(0);  
        }  
    }  
}
```



---

# Chapter 11: The Common Object File Format (COFF)

The Common Object File Format (COFF)	11-1
Definitions and Conventions	11-3
Sections	11-3
Physical and Virtual Addresses	11-3
Target Machine	11-4
File Header	11-4
Magic Numbers	11-5
Flags	11-5
File Header Declaration	11-5
Optional Header Information	11-6
Standard UNIX System <code>a.out</code> Header	11-7
Optional Header Declaration	11-8
Section Headers	11-9
Flags	11-10
Section Header Declaration	11-11
<b>.bss</b> Section Header	11-12
Sections	11-13
Relocation Information	11-13
Relocation Entry Declaration	11-14
Line Numbers	11-15
Line Number Declaration	11-16
Symbol Table	11-17
Special Symbols	11-19
Inner Blocks	11-20
Symbols and Functions	11-22
Symbol Table Entries	11-23
Auxiliary Table Entries	11-37
String Table	11-45
Access Routines	11-46



---

# The Common Object File Format (COFF)

This section describes the Common Object File Format (COFF) used on AT&T computers with the UNIX operating system. COFF is the format of the output file produced by the assembler, **as**, and the link editor, **ld**.

Some key features of COFF are

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains

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

Figure 11-1 shows the overall structure.

FILE HEADER
Optional Information
Section 1 Header
...
Section <i>n</i> Header
Raw Data for Section 1
...
Raw Data for Section <i>n</i>
Relocation Info for Sect. 1
...
Relocation Info for Sect. <i>n</i>
Line Numbers for Sect. 1
...
Line Numbers for Sect. <i>n</i>
SYMBOL TABLE
STRING TABLE

Figure 11-1: Object File Format 

---

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **-s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the **strip** command. The line number information does not appear unless the program is compiled with the **-g** option of the **cc** command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

### Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

#### Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the UNIX operating system loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

#### NOTE:

It is a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

#### Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; but in all versions of COFF on UNIX systems, the physical address is equivalent to the virtual address.

## Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer with the intent of creating an object file that can be executed on another computer. The term target machine refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the exact same computer on which the object file is being created.

## File Header

The file header contains the 20 bytes of information shown in Figure 11-2. The last 2 bytes are flags that are used by `ld` and object file utilities.

Bytes	Declaration	Name	Description
0-1	<b>unsigned short</b>	<b>f_magic</b>	Magic number
2-3	<b>unsigned short</b>	<b>f_nscns</b>	Number of sections
4-7	<b>long int</b>	<b>f_timdat</b>	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	<b>long int</b>	<b>f_symprtr</b>	File pointer containing the starting address of the symbol table
12-15	<b>long int</b>	<b>f_nsyms</b>	Number of entries in the symbol table
16-17	<b>unsigned short</b>	<b>f_opthdr</b>	Number of bytes in the optional header
18-19	<b>unsigned short</b>	<b>f_flags</b>	Flags (see Figure 11-3)

Figure 11-2: File Header Contents



### Magic Numbers

The magic number specifies the target machine on which the object file is executable.

### Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file **filehdr.h**, and are shown in Figure 11-3.

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR32W	0001000	32 bit word
F_BM32B	0020000	32100 required
F_BM32MAU	0040000	MAU required

Figure 11-3: File Header Flags (3B2 Computer)

---

### File Header Declaration

The C structure declaration for the file header is given in Figure 11-4. This declaration may be found in the header file **filehdr.h**.

```
struct filehdr
{
    unsigned short  f_magic;    /* magic number */
    unsigned short  f_nscns;    /* number of section */

    long            f_timdat;    /* time and date stamp */

    long            f_symptr;    /* file ptr to symbol
                                table */

    long            f_nsyms;    /* number entries in the
                                symbol table */

    unsigned short  f_opthdr;    /* size of optional header */

    unsigned short  f_flags;    /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

Figure 11-4: File Header Declaration

## Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field `f_opthdr`.

### Standard UNIX System a.out Header

By default, files produced by the link editor for a UNIX system always have a standard UNIX system **a.out** header in the optional header field. The UNIX system **a.out** header is 28 bytes. The fields of the optional header are described in Figure 11-5.

Bytes	Declaration	Name	Description
0-1	<b>short</b>	<b>magic</b>	Magic number
2-3	<b>short</b>	<b>vstamp</b>	Version stamp
4-7	<b>long int</b>	<b>tsize</b>	Size of text in bytes
8-11	<b>long int</b>	<b>dsize</b>	Size of initialized data in bytes
12-15	<b>long int</b>	<b>bsize</b>	Size of uninitialized data in bytes
16-19	<b>long int</b>	<b>entry</b>	Entry point
20-23	<b>long int</b>	<b>text_start</b>	Base address of text
24-27	<b>long int</b>	<b>data_start</b>	Base address of data

Figure 11-5: Optional Header Contents (3B2, 3B5, 3B15 Computers)

---

Whereas, the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the 3B2/3B5/3B15 UNIX operating system are given in Figure 11-6.

Value	Meaning
0407	The text segment is not write-protected or sharable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write protected.
0413	Text and data segments are aligned within <b>a.out</b> so it can be directly paged.

Figure 11-6: UNIX System Magic Numbers (3B2, 3B5, 3B15 Computers)

### Optional Header Declaration

The C language structure declaration currently used for the UNIX system **a.out** file header is given in Figure 11-7. This declaration may be found in the header file **aouthdr.h**.

```
typedef struct aouthdr
{
    short    magic;    /* magic number */
    short    vstamp;  /* version stamp */
    long     tsize;    /* text size in bytes, padded */
                    /* to full word boundary */

    long     dsize;    /* initialized data size */

    long     bsize;    /* uninitialized data size */

    long     entry;    /* entry point */

    long     text_start; /* base of text for this file */

    long     data_start /* base of data for this file */

} AOUTHDR;
```

Figure 11-7: **aouthdr** Declaration

---

## Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 11-8.

Bytes	Declaration	Name	Description
0-7	<b>char</b>	<b>s_name</b>	8-character null padded section name
8-11	<b>long int</b>	<b>s_paddr</b>	Physical address of section
12-15	<b>long int</b>	<b>s_vaddr</b>	Virtual address of section
16-19	<b>long int</b>	<b>s_size</b>	Section size in bytes
20-23	<b>long int</b>	<b>s_scnptr</b>	File pointer to raw data
24-27	<b>long int</b>	<b>s_relptr</b>	File pointer to relocation entries
28-31	<b>long int</b>	<b>s_innoptr</b>	File pointer to line number entries
32-33	<b>unsigned short</b>	<b>s_nreloc</b>	Number of relocation entries
34-35	<b>unsigned short</b>	<b>s_nlnno</b>	Number of line number entries
36-39	<b>long int</b>	<b>s_flags</b>	Flags (see Figure 11-9)

Figure 11-8: Section Header Contents

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the UNIX system function **fseek(3S)**.

### Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 11-9.

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For <code>.lib</code> section (treated like STYP_INFO)

Figure 11-9: Section Header Flags

---

### Section Header Declaration

The C structure declaration for the section headers is described in Figure 11-10. This declaration may be found in the header file `scnhdr.h`.

```
struct scnhdr
{
  char   s_name[8];    /* section name */
  long   s_paddr;     /* physical address */
  long   s_vaddr;     /* virtual address */
  long   s_size;      /* section size */
  long   s_scnptr;    /* file ptr to section raw data */

  long   s_relptr;    /* file ptr to relocation */

  long   s_lnnoptr;   /* file ptr to line number */

  unsigned short s_nreloc; /*number of relocation entries*/
  unsigned short s_nlnno; /*number of line number entries*/

  long   s_flags;     /* flags */
};

#define SCNHDR struct scnhdr
#define SCNHSZ sizeof(SCNHDR)
```

Figure 11-10: Section Header Declaration

---

### **.bss Section Header**

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0. The same is true of the STYP\_NOLOAD and STYP\_DSECT sections.



## Sections

Figure 11-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor **SECTIONS** directives (see Chapter 12) allow users to, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections

If no **SECTIONS** directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a **.text** section, are linked together the output object file contains a single **.text** section made up of the combined input **.text** sections.

## Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 11-11.

Bytes	Declaration	Name	Description
0-3	<b>long int</b>	<b>r_vaddr</b>	(Virtual) address of reference
4-7	<b>long int</b>	<b>r_symndx</b>	Symbol table index
8-9	<b>unsigned short</b>	<b>r_type</b>	Relocation type

Figure 11-11: Relocation Section Contents

---

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Figure 11-12.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address.
R_DIR32S	012	Direct 32-bit reference to the symbol's virtual address, with the 32-bit value stored in the reverse order in the object file.

Figure 11-12: Relocation Types (3B2, 3B5, 3B15 Computers)

### Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 11-13. This declaration may be found in the header file `reloc.h`.

```
struct reloc
{
    long    r_vaddr;    /* virtual address of reference */
    long    r_symndx;   /* index into symbol table */
    unsigned short  r_type;    /* relocation type */
};

#define RELOC    struct reloc

#define RELSZ    10
```

Figure 11-13: Relocation Entry Declaration

---

## Line Numbers

When invoked with the **-g** option, the **cc**, and **f77** commands cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like **sdb**. All line numbers in a section are grouped by function as shown in Figure 11-14.

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

Figure 11-14: Line Number Grouping

---

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function, and appear in increasing order of address.

### Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 11-15.

```
struct lineno
{
    union
    {
        long    l_symndx;    /* symtbl index of func name */
        long    l_paddr;    /* paddr of line number */
    } l_addr;
    unsigned short  l_lno;    /* line number */
};

#define LINENO      struct lineno
#define LINESZ      6
```

Figure 11-15: Line Number Entry Declaration

---

## Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 11-16.

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
...
statics
...
filename 2
function 1
local symbols for function 1
...
statics
...
defined global symbols
undefined global symbols

Figure 11-16: COFF Symbol Table

---

The word `statics` in Figure 11-16 means symbols defined with the C language storage class `static` outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

## Special Symbols

The symbol table contains some special symbols that are generated by **as**, and other tools. These symbols are given in Figure 11-17.

Symbol	Meaning
<b>.file</b>	filename
<b>.text</b>	address of <b>.text</b> section
<b>.data</b>	address of <b>.data</b> section
<b>.bss</b>	address of <b>.bss</b> section
<b>.bb</b>	address of start of inner block
<b>.eb</b>	address of end of inner block
<b>.bf</b>	address of start of function
<b>.ef</b>	address of end of function
<b>.target</b>	pointer to the structure or union returned by a function
<b>.xfake</b>	dummy tag name for structure, union, or enumeration
<b>.eos</b>	end of members of structure, union, or enumeration
<b>etext</b>	next available address after the end of the output section <b>.text</b>
<b>edata</b>	next available address after the end of the output section <b>.data</b>
<b>end</b>	next available address after the end of the output section <b>.bss</b>

Figure 11-17: Special Symbols in the Symbol Table

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.xfake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is `.xfake`, where `x` is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are `.0fake`, `.1fake`, and `.2fake`. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

### Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces, `{`, and `}`. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol, `.bb`, is put in the symbol table immediately before the first local symbol of that block. Also a special symbol, `.eb`, is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 11-18.

```
.bb  
local symbols  
for that block  
.eb
```

Figure 11-18: Special Symbols (`.bb` and `.eb`)

---

**Because inner blocks can be nested by several levels, the `.bb-.eb` pairs and associated symbols may also be nested. See Figure 11-19.**



```
{
    int i;                /* block 1 */
    char c;
    ...
    {
        long a;          /* block 2 */
        ...
        {
            int x;       /* block 3 */
            ....
        }                /* block 3 */
    }                    /* block 2 */
    {
        long i;          /* block 4 */
        ...
    }                    /* block 4 */
}                        /* block 1 */
```

Figure 11-19: Nested blocks

---

The symbol table would look like Figure 11-20.

<b>.bb</b> for block 1
i
c
<b>.bb</b> for block 2
a
<b>.bb</b> for block 3
x
<b>.eb</b> for block 3
<b>.eb</b> for block 2
<b>.bb</b> for block 4
i
<b>.eb</b> for block 4
<b>.eb</b> for block 1

Figure 11-20: Example of the Symbol Table 

---

### Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 11-21.

function name
<b>.bf</b>
local symbol
<b>.ef</b>

Figure 11-21: Symbols for Functions 

---

### Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 11-22. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Declaration	Name	Description
0-7	(see text below)	<code>_n</code>	These 8 bytes contain either a symbol name or an index to a symbol
8-11	<b>long int</b>	<code>n_value</code>	Symbol value; storage class dependent
12-13	<b>short</b>	<code>n_scnum</code>	Section number of symbol
14-15	<b>unsigned short</b>	<code>n_type</code>	Basic and derived type specification
16	<b>char</b>	<code>n_sclass</code>	Storage class of symbol
17	<b>char</b>	<code>n_numaux</code>	Number of auxiliary entries

Figure 11-22: Symbol Table Entry Format

---

### Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 11-23.

Bytes	Declaration	Name	Description
0-7	<b>char</b>	<b>n_name</b>	8-character null-padded symbol name
0-3	<b>long</b>	<b>n_zeroes</b>	Zero in this field indicates the name is in the string table
4-7	<b>long</b>	<b>n_offset</b>	Offset of the name in the string table

Figure 11-23: Name Field

---

Special symbols generated by the C Compilation System are discussed above in "Special Symbols."

**Storage Classes**

The storage class field has one of the values described in Figure 11-24. These **#define**'s may be found in the header file **storclass.h**.

Mnemonic	Value	Storage Class
C_EFCN	-1	physical end of a function
C_NULL	0	-
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field

Figure 11-24: Storage Classes (Sheet 1 of 2)

---

Mnemonic	Value	Storage Class
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	filename
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

Figure 11-24: Storage Classes (Sheet 2 of 2)

All of these storage classes except for C\_ALIAS and C\_HIDDEN are generated by the **cc** or **as** commands. The compress utility, **cpres**, generates the C\_ALIAS mnemonic. This utility (described in the *User's Reference Manual*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C\_HIDDEN is not used by any UNIX system tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C\_EFCN, C\_EXTDEF, C\_ULABEL, C\_USTATIC, and C\_LINE.

**Storage Classes for Special Symbols**

Some special symbols are restricted to certain storage classes. They are given in Figure 11-25.

<b>Special Symbol</b>	<b>Storage Class</b>
<b>.file</b>	C_FILE
<b>.bb</b>	C_BLOCK
<b>.eb</b>	C_BLOCK
<b>.bf</b>	C_FCN
<b>.ef</b>	C_FCN
<b>.target</b>	C_AUTO
<b>.xfake</b>	C_STRTAG, C_UNTAG, C_ENTAG
<b>.eos</b>	C_EOS
<b>.text</b>	C_STAT
<b>.data</b>	C_STAT
<b>.bss</b>	C_STAT

Figure 11-25: Storage Class by Special Symbols

---

Also some storage classes are used only for certain special symbols. They are summarized in Figure 11-26.

<b>Storage Class</b>	<b>Special Symbol</b>
C_BLOCK	<b>.bb, .eb</b>
C_FCN	<b>.bf, .ef</b>
C_EOS	<b>.eos</b>
C_FILE	<b>.file</b>

Figure 11-26: Restricted Storage Classes

---

**Symbol Value Field**

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Figure 11-27.

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

Figure 11-27: Storage Class and Value

---

If a symbol has storage class C\_FILE, the value of that symbol equals the symbol table entry index of the next .file symbol. That is, the .file entries form a one-way linked list in the symbol table. If there are no more .file entries in the symbol table, the value of the symbol is the index of the first global symbol.



Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

**Section Number Field**

Section numbers are listed in Figure 11-28.

<b>Mnemonic</b>	<b>Section Number</b>	<b>Meaning</b>
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

Figure 11-28: Section Number

---

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and `.eos` symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (i.e., FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0 and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the `.bss` section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

**Section Numbers and Storage Classes**

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 11-29.

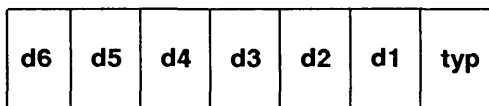
<b>Storage Class</b>	<b>Section Number</b>
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARAM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Figure 11-29: Section Number and Storage Class

---

***Type Entry***

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is



Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Figure 11-30.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_VOID	1	void
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Figure 11-30: Fundamental Types

---

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Figure 11-31.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Figure 11-31: Derived Types

---

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

### ***Type Entries and Storage Classes***

Figure 11-32 shows the type entries that are legal for each storage class.

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARAM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Figure 11-32: Type Entries by Storage Class

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

***Structure for Symbol Table Entries***

The C language structure declaration for the symbol table entry is given in Figure 11-33. This declaration may be found in the header file **syms.h**.

```
struct syment
{
    union
    {
        char        _n_name[SYMNMLEN];    /* symbol name*/
        struct
        {
            long    _n_zeroes;    /* symbol name */

            long    _n_offset;    /* location in string table */
        } _n_n;
        char        *_n_nptr[2]; /* allows overlaying */
    } _n;
    unsigned long  n_value; /* value of symbol */

    short          n_scnum; /* section number */

    unsigned short n_type; /* type and derived */

    char           n_sclass; /* storage class */

    char           n_numaux; /* number of aux entries */
};

#define n_name        _n._n_name
#define n_zeroes     _n._n_n._n_zeroes
#define n_offset     _n._n_n._n_offset
#define n_nptr       _n._n_nptr[1]

#define SYMNMLEN    8
#define SYMESZ      18    /* size of a symbol table entry */
```

Figure 11-33: Symbol Table Entry Declaration



### Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 11-34.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
<b>.file</b>	C_FILE	DT_NON	T_NULL	filename
<b>.text,.data, .bss</b>	C_STAT	DT_NON	T_NULL	section
<i>tagname</i>	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
<b>.eos</b>	C_EOS	DT_NON	T_NULL	end of structure
<i>fcname</i>	C_EXT C_STAT	DT_FCN	(Note 1)	function
<i>arrname</i>	(Note 2)	DT_ARY	(Note 1)	array
<b>.bb,.eb</b>	C_BLOCK	DT_NON	T_NULL	beginning and end of block
<b>.bf,.ef</b>	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Figure 11-34: Auxiliary Symbol Table Entries

Notes to Figure 11-34:

1. Any except T\_MOE.
2. C\_AUTO, C\_STAT, C\_MOS, C\_MOU, C\_TPDEF.

In Figure 11-34, *tagname* means any symbol name including the special symbol *xfake*, and *fname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Figure 11-34 should have a union format in its auxiliary entry.

**NOTE:** It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available, and should be obtained from the `n_numaux` field in the symbol table.

### **Filenames**

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes 0 through 13. The remaining bytes are 0.

### **Sections**

The auxiliary table entries for sections have the format as shown in Figure 11-35.

<b>Bytes</b>	<b>Declaration</b>	<b>Name</b>	<b>Description</b>
0-3	<b>long int</b>	<b>x_scnlen</b>	section length
4-5	<b>unsigned short</b>	<b>x_nreloc</b>	number of relocation entries
6-7	<b>unsigned short</b>	<b>x_nlinno</b>	number of line numbers
8-17	-	-	unused (filled with zeroes)

Figure 11-35: Format for Auxiliary Table Entries for Sections

---

### **Tag Names**

The auxiliary table entries for tag names have the format shown in Figure 11-36.

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeroes)
6-7	<b>unsigned short</b>	<b>x_size</b>	size of structure, union, and enumeration
8-11	-	-	unused (filled with zeroes)
12-15	<b>long int</b>	<b>x_endndx</b>	index of next entry beyond this structure, union, or enumeration
16-17	-	-	unused (filled with zeroes)

Figure 11-36: Tag Names Table Entries

---

### *End of Structures*

The auxiliary table entries for the end of structures have the format shown in Figure 11-37:

Bytes	Declaration	Name	Description
0-3	<b>long int</b>	<b>x_tagndx</b>	tag index
4-5	-	-	unused (filled with zeroes)
6-7	<b>unsigned short</b>	<b>x_size</b>	size of structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Figure 11-37: Table Entries for End of Structures

---

### *Functions*

The auxiliary table entries for functions have the format shown in Figure 11-38:

Bytes	Declaration	Name	Description
0-3	<b>long int</b>	<b>x_tagndx</b>	tag index
4-7	<b>long int</b>	<b>x_fsize</b>	size of function (in bytes)
8-11	<b>long int</b>	<b>x_innoptr</b>	file pointer to line number
12-15	<b>long int</b>	<b>x_endndx</b>	index of next entry beyond this point
16-17	<b>unsigned short</b>	<b>x_tvndx</b>	index of the function's address in the transfer vector table (not used in UNIX system)

Figure 11-38: Table Entries for Functions

**Arrays**

The auxiliary table entries for arrays have the format shown in Figure 11-39. Defining arrays having more than four dimensions produces a warning message.

Bytes	Declaration	Name	Description
0-3	<b>long int</b>	<b>x_tagndx</b>	tag index
4-5	<b>unsigned short</b>	<b>x_inno</b>	line number of declaration
6-7	<b>unsigned short</b>	<b>x_size</b>	size of array
8-9	<b>unsigned short</b>	<b>x_dimen[0]</b>	first dimension
10-11	<b>unsigned short</b>	<b>x_dimen[1]</b>	second dimension
12-13	<b>unsigned short</b>	<b>x_dimen[2]</b>	third dimension
14-15	<b>unsigned short</b>	<b>x_dimen[3]</b>	fourth dimension
16-17	-	-	unused (filled with zeroes)

Figure 11-39: Table Entries for Arrays

***End of Blocks and Functions***

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 11-40:

<b>Bytes</b>	<b>Declaration</b>	<b>Name</b>	<b>Description</b>
0-3	-	-	unused (filled with zeroes)
4-5	<b>unsigned short</b>	<b>x_inno</b>	C-source line number
6-17	-	-	unused (filled with zeroes)

Figure 11-40: End of Block and Function Entries

---

***Beginning of Blocks and Functions***

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 11-41:

<b>Bytes</b>	<b>Declaration</b>	<b>Name</b>	<b>Description</b>
0-3	-	-	unused (filled with zeroes)
4-5	<b>unsigned short</b>	<b>x_inno</b>	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	<b>long int</b>	<b>x_endndx</b>	index of next entry past this block
16-17	-	-	unused (filled with zeroes)

Figure 11-41: Format for Beginning of Block and Function

---

***Names Related to Structures, Unions, and Enumerations***

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Figure 11-42:

Bytes	Declaration	Name	Description
0-3	<b>long int</b>	<b>x_tagndx</b>	tag index
4-5	-	-	unused (filled with zeroes)
6-7	<b>unsigned short</b>	<b>x_size</b>	size of the structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Figure 11-42: Entries for Structures, Unions, and Enumerations

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;
struct people
{
    char name[20];
    long id;
};
typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table but symbol STUDENT will not because it is a forward reference to a structure.

### ***Auxiliary Entry Declaration***

The C language structure declaration for an auxiliary symbol table entry is given in Figure 11-43. This declaration may be found in the header file **syms.h**.

```
union auxent
{
    struct
    {
        long    x_tagndx;
        union
        {
            struct
            {
                unsigned short  x_lno;
                unsigned short  x_size;
            } x_lnsz;
            long    x_fsize;
        } x_misc;
        union
        {
            struct
```

.  
.  
.

Figure 11-43: Auxiliary Symbol Table Entry (Sheet 1 of 2)

---

```

.
.
.
        {
            long    x_lnnoptr;
            long    x_endndx;
        } x_fcn;
    struct
    {
        unsigned short    x_dimen[DIMNUM];
    } x_ary;
    } x_fcenary;
    unsigned short    x_tvndx;
} x_sym;
struct
{
    char    x_fname[FILNMLEN];
} x_file;
struct
{
    long    x_scnlen;
    unsigned short    x_nreloc;
    unsigned short    x_nlinno;
} x_scn;
struct
{
    long    x_tvfill;
    unsigned short    x_tvlen;
    unsigned short    x_tvran[2];
} x_tv;
}
#define FILNMLEN 14
#define DIMNUM 4
#define AUXENT union auxent
#define AUXESZ 18

```

Figure 11-43: Auxiliary Symbol Table Entry (Sheet 2 of 2)

---



## String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer than eight characters, **long\_name\_1** and **another\_one**) the string table has the format as shown in Figure 11-44:

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

Figure 11-44: String Table

---

The index of **long\_name\_1** in the string table is 4 and the index of **another\_one** is 16.

### Access Routines

UNIX system releases contain a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of the *Programmer's Reference Manual*. A summary of what is available can be found in the *Programmer's Reference Manual* under **ldfcn(4)**.

---

# Chapter 12: The Link Editor

The Link Editor	12-1
Memory Configuration	12-1
Sections	12-2
Addresses	12-2
Binding	12-2
Object File	12-3
Link Editor Command Language	12-4
Expressions	12-4
Assignment Statements	12-5
Specifying a Memory Configuration	12-7
Section Definition Directives	12-9
File Specifications	12-10
Load a Section at a Specified Address	12-11
Aligning an Output Section	12-12
Grouping Sections Together	12-13
Creating Holes Within Output Sections	12-16
Creating and Defining Symbols at Link-Edit Time	12-18
Allocating a Section Into Named Memory	12-20
Initialized Section Holes or <b>.bss</b> Sections	12-20
Notes and Special Considerations	12-23
Changing the Entry Point	12-23
Use of Archive Libraries	12-23
Dealing With Holes in Physical Memory	12-26
Allocation Algorithm	12-27
Incremental Link Editing	12-28
DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections	12-30

## Table of Contents

---

Output File Blocking	12-32
Nonrelocatable Input Files	12-32
Syntax Diagram for Input Directives	12-34

---

# The Link Editor

In Chapter 2 there was a discussion of link editor command line options (some of which may also be provided on the `cc(1)` command line). This chapter contains information on the Link Editor Command Language.

The command language enables you to

- specify the memory configuration of the target machine
- combine the sections of an object file in arrangements other than the default
- bind sections to specific addresses or within specific portions of memory
- define or redefine global symbols

Under most normal circumstances there is no compelling need to have such tight control over object files and where they are located in memory. When you do need to be very precise in controlling the link editor output, you do it by means of the command language.

Link editor command language directives are passed in a file named on the `ld(1)` command line. Any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives. The following paragraphs define terms and describe conditions with which you need to be familiar before you begin to use the command language.

## Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K-1 are then not configured. Unconfigured memory is treated as reserved or unusable by `ld(1)`. Nothing can ever be linked into

unconfigured memory. Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) illegal or nonexistent with respect to the linking process. Memory configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc. are with respect to the configured sections of the address space.

## Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

## Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

## Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the `ld(1)` command language.

## Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by **ld(1)**. **ld(1)** accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to **ld(1)** can also be absolute files.

Files produced from the compilation system may contain, among others, sections called **.text** and **.data**. The **.text** section contains the instruction text (executable instructions), **.data** contains initialized data variables. For example, if a C program contained the global (i.e., not inside a function) declaration

```
int i = 100;
```

and the assignment

```
i = 0;
```

then compiled code from the C assignment is stored in **.text**, and the variable **i** is located in **.data**.

---

# Link Editor Command Language

## Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 12-2, "Syntax Diagram for Input Directives.") Constants are as in C with a number recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers's. Symbol names may contain uppercase or lowercase letters, digits, and the underscore, `_`. Symbols within an expression have the value of the address of the symbol only. `ld(1)` does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

`ld(1)` uses a `lex`-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names reserved and unavailable as symbol names or section names:

<b>ADDR</b>	<b>BLOCK</b>	<b>GROUP</b>	<b>NEXT</b>	<b>RANGE</b>	<b>SPARE</b>
<b>ALIGN</b>	<b>COMMON</b>	<b>INFO</b>	<b>NOLOAD</b>	<b>REGIONS</b>	<b>PHY</b>
<b>ASSIGN</b>	<b>COPY</b>	<b>LENGTH</b>	<b>ORIGIN</b>	<b>SECTIONS</b>	<b>TV</b>
<b>BIND</b>	<b>DSECT</b>	<b>MEMORY</b>	<b>OVERLAY</b>	<b>SIZEOF</b>	

<b>addr</b>	<b>block</b>	<b>length</b>	<b>origin</b>	<b>sizeof</b>
<b>align</b>	<b>group</b>	<b>next</b>	<b>phy</b>	<b>spare</b>
<b>assign</b>	<b>l</b>	<b>o</b>	<b>range</b>	
<b>bind</b>	<b>len</b>	<b>org</b>	<b>s</b>	

The operators that are supported, in order of precedence from high to low, are shown in Figure 12-1:



symbol
! - - (UNARY Minus)
* / %
+ - (BINARY Minus)
>> <<
== != > < <= >=
&
&&
= += -= *= /=

Figure 12-1: Operator Symbols

---

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

## Assignment Statements

External symbols may be defined and assigned addresses via the assignment statement. The syntax of the assignment statement is

```
symbol = expression;
```

or

```
symbol op= expression;
```

where *op* is one of the operators +, -, \*, or /. Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol

address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to **ld(1)**.

Assignment statements are normally placed outside the scope of section-definition directives (see "Section Definition Directives" under "Link Editor Command Language"). However, there exists a special symbol, called **dot**, `.`, that can occur only within a section-definition directive. This symbol refers to the current address of **ld(1)**'s location counter. Thus, assignment expressions involving `.` are evaluated during the allocation phase of **ld(1)**. Assigning a value to the `.` symbol within a section-definition directive can increment (but not decrement) **ld(1)**'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of the `.` symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

**align** is provided as a shorthand notation to allow alignment of a symbol to an  $n$ -byte boundary within an output section, where  $n$  is a power of 2. For example, the expression

```
align(n)
```

is equivalent to

```
(. + n - 1) &~ (n - 1)
```

**SIZEOF** and **ADDR** are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside of section directives.

Link editor expressions may have either an absolute or a relocatable value. When **ld(1)** creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.
- The difference of two relocatable symbols from the same section is absolute.

- All other expressions are combinations of the above.

## Specifying a Memory Configuration

MEMORY directives are used to specify

1. The total size of the virtual space of the target machine.
2. The configured and unconfigured areas of the virtual space.

If no directives are supplied, **ld(1)** assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters \$, ., or \_. Names of memory ranges are used by **ld(1)** only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld(1)**'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory.

As an option on the MEMORY directive, attributes may be associated with a named memory area. In future releases this may be used to provide error checking. Currently, error checking of this type is not implemented.

The attributes currently accepted are

1. R : readable memory
2. W : writable memory
3. X : executable, i.e., instructions may reside in this memory
4. I : initializable, i.e., stack areas are typically not initialized

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

The syntax of the MEMORY directive is

```
MEMORY
{
    name1 (attr) :    origin = n1, length = n2
    name2 (attr) :    origin = n3, length = n4
    etc.
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype. The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, **ld(1)** can be told that memory is configured in some manner other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this.

```
MEMORY
{
    valid : org = 0x10000, len = 0xFE0000
}
```

## Section Definition Directives

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections s1 and s2 and the other containing sections s3 and s4, the output object file contains the four sections s1, s2, s3, and s4. The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is

```
SECTIONS
{
    secname1 :
    {
        file_specifications,
        assignment_statements
    }
    secname2 :
    {
        file_specifications,
        assignment_statements
    }
    etc.
}
```

The various types of section definition directives are discussed in the remainder of this section.

### File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

```
filename ( secname )
```

or

```
filename ( secnam1 secnam2 . . . )
```

Sections of an input file are separated either by white space or commas as are the file specifications themselves.

```
filename [COMMON]
```

may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

If a file name appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section. For example,

```
SECTIONS
{
    outsec1:
    {
        file1.o (sec1)
        file2.o
        file3.o (sec1, sec2)
    }
}
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be

1. section **sec1** from file **file1.o**
2. all sections from **file2.o**, in the order they appear in the file
3. section **sec1** from file **file3.o**, and then section **sec2** from file **file3.o**

If there are any additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in **file1.o** or **file3.o**, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The code

```
*(secname)
```

may be used to indicate all previously unallocated input sections of the given name, regardless of what input file they are contained in.

### Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an **ld(1)** option as shown in the following **SECTIONS** directive example:

```
SECTIONS
{
    outsec addr:
    {
        . . .
    }
    etc.
}
```

The *addr* is the bonding address expressed as a C constant. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld(1)** issues an

appropriate error message. *addr* may also be the word BIND, followed by a parenthesized expression. The expression may use the pseudo-functions SIZEOF, ADDR or NEXT. NEXT accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; SIZEOF and ADDR accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to **ld(1)** in any particular order, unless SIZEOF or ADDR is used.

**ld(1)** does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. **ld(1)** directives can be used to force a section to start on an odd byte boundary although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld(1)** issues a warning message.

### Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option

ALIGN(*n*)

is equivalent to specifying a bonding address of

$( \cdot + n - 1 ) \& \sim ( n - 1 )$

For example



```
SECTIONS
{
    outsec  ALIGN(0x20000) :
    {
        . . .
    }
    etc.
}
```

The output section **outsec** is not bound to any given address but is placed at some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

### Grouping Sections Together

The default allocation algorithm for **ld(1)**

1. Links all input **.init** sections together, followed by **.text** sections, into one output section. This output section is called **.text** and is bound to an address of 0x0 plus the size of all headers in the output file.
2. Links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine dependent constant plus a number dependent on the size of headers and text.
3. Links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated so as to immediately follow the output section **.data**. Note that the output section **.bss** is not given any particular address alignment.

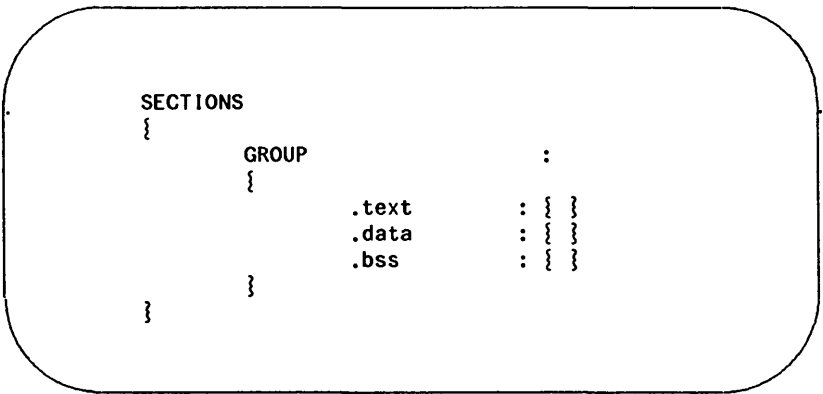
Specifying any SECTIONS directives results in this default allocation not being performed. Rather than relying on the `ld(1)` default algorithm, if you are manipulating COFF files, the one certain way of determining address and order information is to take it from the file and section headers. The default allocation of `ld(1)` is equivalent to supplying the following directive:

```

SECTIONS
{
    .text sizeof_headers : { *(.init) *(.text) }
    GROUP BIND( NEXT(align_value) +
                ((SIZEOF(.text) + ADDR(.text)) % 0x2000) ) :
    {
        .data      : { }
        .bss       : { }
    }
}
    
```

where *align\_value* is a machine dependent constant. The `GROUP` command ensures that the two output sections, `.data` and `.bss`, are allocated (e.g., grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If `.text`, `.data`, and `.bss` are to be placed in the same segment, the following SECTIONS directive is used:



Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the **GROUP** directive. To bind to **0xC0000**, use

```
GROUP 0xC0000 : {
```

To align to **0x10000**, use

```
GROUP ALIGN(0x10000) : {
```

With this addition, first the output section **.text** is bound at **0xC0000** (or is aligned to **0x10000**); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the **GROUP** directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
    .text    : { }
    .data ALIGN(0x20000) : { }
    .bss     : { }
}
```

The **.text** section starts at virtual address 0x0 (if it is in configured memory) and the **.data** section at a virtual address aligned to 0x20000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section. The order in which output sections are defined to **ld(1)** cannot be used to force a certain allocation order in the output file.

### Creating Holes Within Output Sections

The special symbol dot, **.**, appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, **.** causes **ld(1)**'s location counter to be incremented or reset and a hole left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the **-f** option in "Using the Link Editor" and the discussion of filling holes in "Initialized Section Holes" or **.bss** Sections." in this chapter.

Consider the following section definition:

```
outsec:
{
    . += 0x1000;
    f1.o (.text)
    . += 0x100;
    f2.o (.text)
    . = align (4);
    f3.o (.text)
}
```

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section **f1.o (.text)** is linked after this hole.
2. The **.text** section of input file **f2.o** begins at 0x100 bytes following the end of **f1.o (.text)**.
3. The **.text** section of **f3.o** is linked to start at the next full word boundary following the **.text** section of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld(1)** treats the output section as if it began at address zero. As a result, if, in the above example, **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address—even though **f3.o (.text)** is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

It should be noted that the assembler, **as**, always pads the sections it generates to a full word length making explicit alignment specifications unnecessary. This also holds true for the compiler.

Expressions that decrement `.` are illegal. For example, subtracting a value from the location counter is not allowed since overwrites are not allowed. The most common operators in expressions that assign a value to `.` are `+=` and `align`.

### Creating and Defining Symbols at Link-Edit Time

The assignment instruction of `ld(1)` can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. Use of `.` to adjust `ld(1)`'s location counter during allocation.
2. Use of `.` to assign an allocation-dependent value to a symbol.
3. Assigning an allocation-independent value to a symbol.

Case 1) has already been discussed in the previous section.

Case 2) provides a means to assign addresses (known only after allocation) to symbols. For example,

```
SECTIONS
{
    outsc1: {...}
    outsc2:
    {
        file1.o (s1)
        s2_start = . ;
        file2.o (s2)
        s2_end = . - 1;
    }
}
```

The symbol `s2_start` is defined to be the address of `file2.o(s2)`, and `s2_end` is the address of the last byte of `file2.o(s2)`.

Consider the following example:

```
SECTIONS
{
    outsc1:
    {
        file1.o (.data)
        mark = .;
        . += 4;
        file2.o (.data)
    }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file1.o**'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving **.** must appear within **SECTIONS** definitions since they are evaluated during allocation. Assignment instructions that do not involve **.** can appear within **SECTIONS** definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. The **ld(1)** issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

## Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a MEMORY directive). (The > notation is borrowed from the UNIX system concept of redirected output.) For example,

```

MEMORY
{
    mem1:          o=0x000000    l=0x10000
    mem2 (RW):     o=0x020000    l=0x40000
    mem3 (RW):     o=0x070000    l=0x40000
    mem1:          o=0x120000    l=0x04000
}

SECTIONS
{
    outsec1: { f1.o(.data) } > mem1
    outsec2: { f2.o(.data) } > mem3
}

```

This directs **ld(1)** to place **outsec1** anywhere within the memory area named **mem1** (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

## Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Creating Holes within Output Sections"), **ld(1)** normally puts out bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling



the `.o` file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized `.bss` section, if part of such a section is initialized, then the entire section is initialized. In other words, if a `.bss` section is to be combined with a `.text` or `.data` section (both of which are initialized) or if part of an output `.bss` section is to be initialized, then one of the following will hold:

1. Explicit initialization options must be used to initialize all `.bss` sections in the output section.
2. `ld(1)` will use the default fill value to initialize all `.bss` sections in the output section.

Consider the following `ld(1)` ifile:

```

SECTIONS
{
    sec1:
    {
        f1.o
        . =+ 0x200;
        f2.o (.text)
    } = 0xDFFF
    sec2:
    {
        f1.o (.bss)
        f2.o (.bss) = 0x1234
    }
    sec3:
    {
        f3.o (.bss)
        . . .
    } = 0xFFFF
    sec4: { f4.o (.bss) }
}

```

In the example above, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2**, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All **.bss** sections within **sec3** as well as all holes are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object file for this section.

---

# Notes and Special Considerations

## Changing the Entry Point

The UNIX system **a.out** optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

1. The value of the symbol specified with the **-e** option, if present, is used.
2. The value of the symbol **\_start**, if present, is used.
3. The value of the symbol **main**, if present, is used.
4. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the **-e** option or by using an assignment instruction in an ifile of the form

```
_start = expression;
```

If **ld(1)** is called through **cc(1)**, a startup routine is automatically linked in. Then, when the program is executed, the routine **exit(1)** is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld(1)** directly or when changing the entry point. The user must supply the startup routine or make sure that the program always calls **exit** rather than falling through the end. Otherwise, the program will dump core.

## Use of Archive Libraries

Each member of an archive library (e.g., **libc.a**) is a complete object file. Archive libraries are created with the **ar(1)** command from object files generated by **cc** or **as**. An archive library is always processed using selective inclusion: only those members that resolve existing undefined-symbol references are taken from the library for link editing. Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

1. There exists a reference to a symbol defined in that member.
2. The reference is found by **ld(1)** prior to the actual scanning of the library.

When a library member is included by searching the library inside a **SECTIONS** directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library outside of a **SECTIONS** directive, all input sections from the library member are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

1. Specific members of a library cannot be referenced explicitly in an ifile.
2. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The **-l** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the **-l** option by simply giving the (full or relative) UNIX system file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld(1)** will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

1. The input files **file1.o** and **file2.o** each contain a reference to the external function **FCN**.

2. Input **file1.o** contains a reference to symbol ABC.
3. Input **file2.o** contains a reference to symbol XYZ.
4. Library **liba.a**, member 0, contains a definition of XYZ.
5. Library **libc.a**, member 0, contains a definition of ABC.
6. Both libraries have a member 1 that defines FCN.

If the **ld(1)** command were entered as

**ld file1.o -la file2.o -lc**

then the FCN references are satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ remains undefined (because the library **liba.a** is searched before **file2.o** is specified).

If the **ld(1)** command were entered as

**ld file1.o file2.o -la -lc**

then the FCN references is satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0. If the **ld(1)** command were entered as

**ld file1.o file2.o -lc -la**

then the FCN references is satisfied by **libc.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0.

The **-u** option is used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

**ld -u rout1 -la**

creates an undefined symbol called **rout1** in **ld(1)**'s global symbol table. If any member of library **liba.a** defines this symbol, it (and perhaps other members as well) is extracted. Without the **-u** option, there would have been no unresolved references or undefined symbols to cause **ld(1)** to search the archive library.

## Dealing With Holes in Physical Memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
    mem1:      o = 0x00000      1 = 0x02000
    mem2:      o = 0x40000      1 = 0x05000
    mem3:      o = 0x20000      1 = 0x10000
}
```

Let the files **f1.o**, **f2.o**, . . . **fn.o** each contain three sections **.text**, **.data**, and **.bss**, and suppose the combined **.text** section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the **.text** output section so **ld(1)** may do allocation. For example,

```
SECTIONS
{
    txt1:
    {
        f1.o (.text)
        f2.o (.text)
        f3.o (.text)
    }
    txt2:
    {
        f4.o (.text)
        f5.o (.text)
        f6.o (.text)
    }
    etc.
}
```

## Allocation Algorithm

An output section is formed either as a result of a `SECTIONS` directive, by combining input sections of the same name, or by combining `.text` and `.init` into `.text`. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. `ld(1)` uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit bonding addresses were specified are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment

taken into consideration.

3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no `SECTIONS` directives are given, then output sections are allocated in the order they appear to `ld(1)`. Otherwise, output sections are allocated in the order they were defined or made known to `ld(1)` into the first available space they fit.

### Incremental Link Editing

As previously mentioned, the output of `ld(1)` can be used as an input file to subsequent `ld(1)` runs providing that the relocation information is retained (`-r` option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application. For example,



Step 1:

**ld -r -o outfile1 ifile1 infile1.o**

```
/* ifile1 */  
SECTIONS  
{  
    ss1:  
    {  
        f1.o  
        f2.o  
        .  
        .  
        .  
        fn.o  
    }  
}
```

Step 2:

**ld -r -o outfile2 ifile2 infile2.o**

```
/* ifile2 */  
SECTIONS  
{  
    ss2:  
    {  
        g1.o  
        g2.o  
        .  
        .  
        .  
        gn.o  
    }  
}
```

Step 3:

**ld -a -o final.out outfile1 outfile2**

By judiciously forming subsystems, applications may achieve a form of incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules

1. Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.
2. All allocation and memory directives, as well as any assignment statements, are included only in the final ld(1) call.

**DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections**

Sections may be given a type in a section definition as shown in the following example:

```
SECTIONS
{
    name1 0x200000 (DSECT)      : { file1.o }
    name2 0x400000 (COPY)      : { file2.o }
    name3 0x600000 (NOLOAD)    : { file3.o }
    name4          (INFO)      : { file4.o }
    name5 0x900000 (OVERLAY)    : { file5.o }
}
```

The DSECT option creates what is called a dummy section. A dummy section has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld(1)**.
2. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not as a DSECT).
4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from **file1.o** are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the **COPY** option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An **INFO** section is the same as a **COPY** section but its purpose is to carry information about the object file whereas the **COPY** section may contain valid text and data. **INFO** sections are usually used to contain file version identification information.

A section with the type of **NOLOAD** differs in only one respect from a normal output section: its text and/or data is not written to the output file. A **NOLOAD** section is allocated virtual space, appears in the memory map, etc.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

### Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct **ld(1)** to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
    .text BLOCK(0x200) : { }
    .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this SECTIONS directive, **ld(1)** assures that each section, **.text** and **.data**, is physically written at a file offset, which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400, and so forth, in the file).

### Nonrelocatable Input Files

If a file produced by **ld(1)** is intended to be used in a subsequent **ld(1)** run, the first **ld(1)** run should have the **-r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld(1)** does not have relocation or symbol table information (perhaps from the action of a **strip(1)** command, or from being link edited without a **-r** option or with a **-s** option), the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met.

1. Each input file must have no unresolved external references.
2. Each input file must be bound to the exact same virtual address as it was bound to in the **ld(1)** run that created it.

**NOTE:**

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld(1)**.

# Syntax Diagram for Input Directives

Directives	Expanded Directives
< ifile > < cmd >	{ < cmd > } < memory > < sections > < assignment > < filename > < flags >
< memory >	MEMORY { < memory_spec >
< memory_spec >	< name > [ < attributes > ] :
< attributes > < origin_spec > < lenth_spec > < origin > < length >	( { R   W   X   I } ) < origin > = < long > < length > = < long > ORIGIN   o   org   origin LENGTH   l   len   length

Figure 12-2: Syntax Diagram for Input Directives (Sheet 1 of 4)

**NOTE:**

Two punctuation symbols, square brackets and curly braces, do double duty in this diagram.

Where the actual symbols, [] and {} are used, they are part of the syntax and must be present when the directive is specified.

Where you see the symbols [ and ] (larger and in bold), it means the material enclosed is optional.

Where you see the symbols { and } (larger and in bold), it means multiple occurrences of the material enclosed are permitted.

Directives	Expanded Directives
<sections>	SECTIONS { {<sec_or_group> } }
<sec_or_group> <group>	<section>   <group>   <library> GROUP <group_options> : { <section_list> } [ <mem_spec> ]
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> : { <statement> } [ <fill> ] [ <mem_spec> ]
<group_options>	[ <addr> ]   [ <align_option> ] [ <block_option> ]
<sec_options>	[ <addr> ]   [ <align_option> ] [ <block_option> ] [ <type_option> ]
<addr>	<long>   <bind> ( <expr> )
<alignoption>	<align> ( <expr> )
<align>	ALIGN   align
<block_option>	<block> ( <long> )
<block>	BLOCK   block
<type_option>	(DSECT)   (NOLOAD)   (COPY)   (INFO)   (OVERLAY)
<fill>	= <long>
<mem_spec>	> <name> > <attributes>
<statement>	<filename> <filename> ( <name_list> )   [COMMON] * ( <name_list> )   [COMMON] <assignment>

Figure 12-2: Syntax Diagram for Input Directives (Sheet 2 of 4)

---

## Syntax Diagram for Input Directives

Directives	Expanded Directives
<name_list>	<section_name> [,] { <section_name> }
<library>	-l <name>
<bind>	BIND   bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name>   .
<assign_op>	=   + =   -=   *=   /= =
<end>	;
<expr>	<expr> <binary_op> <expr>
<binary_op>	<term> *   /   % +   - > >   < < ==   !=   >   <   < =   > = &   &&    
<term>	<long> <name> <align> ( <term> ) ( <expr> ) <unary_op> <term> <phy> ( <lside> ) <sizeof> ( <sectionname> ) <next> ( <long> ) <addr> ( <sectionname> )
<unary_op>	!   -
<phy>	PHY   phy
<sizeof>	SIZEOF   sizeof

Figure 12-2: Syntax Diagram for Input Directives (Sheet 3 of 4)



Directives	Expanded Directives
<p>&lt; next &gt;                      &lt; addr &gt;                      &lt; flags &gt;</p>	<p>NEXT   next                      ADDR   addr                      -e &lt; wht_space &gt; &lt; name &gt;                      -f &lt; wht_space &gt; &lt; long &gt;                      -h &lt; wht_space &gt; &lt; long &gt;                      -l &lt; name &gt;                      -m                      -o &lt; wht_space &gt; &lt; filename &gt;                      -r                      -t                      -u &lt; wht_space &gt; &lt; name &gt;                      -z                      -H                      -L &lt; path_name &gt;                      -M                      -N                      -S                      -V                      -VS &lt; wht_space &gt; &lt; long &gt;                      -x</p>
<p>&lt; name &gt;                      &lt; long &gt;                      &lt; wht_space &gt;                      &lt; filename &gt;</p>	<p>Any valid symbol name                      Any valid long integer constant                      Blanks, tabs, and newlines                      Any valid UNIX operating system filename. This may include a full or partial path name.</p>
<p>&lt; sectionname &gt;</p>	<p>Any valid section name, up to 8 characters</p>
<p>&lt; path_name &gt;</p>	<p>Any valid UNIX operating system path name (full or partial)</p>

Figure 12-2: Syntax Diagram for Input Directives (Sheet 4 of 4)

