Topics in IRIX Programming

Document Number 007-2478-001

CONTRIBUTORS

Written by Arthur Evans, Wendy Ferguson, and Jed Hartman
Edited by Christina Cary
Production by Laura Cooper and Lorrie Williams
Engineering contributions by Ivan Bach, Greg Boyd, Bill Mannell, Huy Nguyen, and Paul Mielke
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

Topics in IRIX Programming
Document Number 007-2478-001

Contents

List of Examples

List of Figures

List of Tables

About This Manual

This manual discusses a few topics of interest to programmers writing applications for the IRIX™ operating system. Topics include inter-process communication, file and record locking, fonts, and internationalization.

What This Manual Contains

This manual contains the following chapters:

- Chapter 1, “Inter-Process Communication,” describes System V and IRIX inter-process communication mechanisms.
- Chapter 2, “File and Record Locking,” describes how to lock and unlock files and parts of files from within a program.
- Chapter 3, “Working with Fonts,” discusses typography and font use on Silicon Graphics computers, and describes the Font Manager library.
- Chapter 4, “Internationalizing Your Application,” explains how to create an application that can be adapted for use in different countries.
- Appendix A, “ISO 3166 Country Names and Abbreviations,” lists country codes for use with internationalization and localization.

For an overview of the IRIX programming environment and tools available for application programming, see *Programming on Silicon Graphics Systems: An Overview.*

What You Should Know Before Reading This Manual

This manual is for anyone who wants to program effectively under the IRIX operating system. We assume you are familiar with the IRIX (or UNIX®) operating system and a programming language such as C.

Suggestions for Further Reading

In addition to this manual, which covers IRIX topics, you may want to refer to other Silicon Graphics manuals that describe compilers and programming languages. The following table lists where you can find this information.

Table In-1 Suggestions for Further Reading

Topic	Document
IRIX programming	Programming on Silicon Graphics Systems: An Overview
Compiling	MIPS Compiling and Performance Tuning Guide
Assembly language	MIPSpro Assembly Language Programmer's Guide
C language	C Language Reference Manual
C++ language	C++ Programming Guide
Fortran language	Fortran77 Programmer's Guide
Pascal language	Pascal Programming Guide and Man Pages
Real-time programming	REACT/Pro Release Notes

You can order a printed manual from Silicon Graphics by calling SGI Direct at 1-800-800-SGI1 (800-7441). Outside the U.S. and Canada, contact your local sales office or distributor.

Silicon Graphics also provides manuals online. To read an online manual after installing it, type **`insight`** or double-click the InSight icon. It's easy to print sections and chapters of the online manuals from InSight.

You may also want to learn more about standard UNIX topics. For UNIX information, consult a computer bookstore or one of the following:

- AT&T. *UNIX System V Release 4 Programmer's Guide*. Englewood Cliffs, NJ: Prentice Hall, 1990
- Levine, Mason, and Brown. *lex & yacc*. Sebastopol. CA: O'Reilly & Associates, Inc., 1992

- Oram and Talbott. *Managing Projects with make*. Sebastopol. CA: O'Reilly & Associates, Inc., 1991

Conventions Used in This Manual

This manual uses these conventions and symbols:

`Courier`	In text, the Courier font represents function names, file names, and keywords. It is also used for command syntax, output, and program listings.
`bold`	Boldface is used along with Courier font to represent user input.
italics	Words in italics represent characters or numerical values that you define. Replace the abbreviation with the defined value. Also, italics are used for manual page names and commands. The section number, in parentheses, follows the name.
[ ]	Brackets enclose optional items.
{ }	Braces enclose two or more items; you must specify at least one of the items.
|	The OR symbol separates two or more optional items.
…	A horizontal ellipsis in a syntax statement indicates that the preceding optional items can appear more than once in succession.
( )	Parentheses enclose entities and must be typed.

The following two examples illustrate the syntax conventions:

`DIMENSION` *`a`*`(`*`d`*`) [,`*`a`*`(`*`d`*`)] …`

indicates that the Fortran keyword DIMENSION must be typed as shown, that the user-defined entity *a(d)* is required, and that one or more of *a(d)* can be specified. The parentheses ( ) enclosing **d** are required.

`{STATIC | AUTOMATIC}` *v* `[,`*v*`]` …

indicates that either the STATIC or AUTOMATIC keyword must be typed as shown, that the user-defined entity *v* is required, and that one or more *v* items can be specified.

Chapter 1

Inter-Process Communication

This chapter describes System V Release 4 and IRIX inter-process communication mechanisms.

Chapter 1

Inter-Process Communication

The term *Inter-Process Communication* (*IPC*) describes any method of sending data from one running process to another. IPC is commonly used to allow processes to cooperate—for instance, to let two subprograms use the same data areas in memory without interfering with each other—or to make data acquired by one process available to others. A variety of IPC mechanisms exist, each intended for a different purpose.

This chapter describes IPC and covers the following topics:

- "Types of Inter-Process Communication Available" covers the types of IPC available on IRIX systems
- "System V IPC" describes System V messages, semaphores, and shared memory.
- "IRIX IPC" explains IRIX shared arenas, spinlocks, semaphores, and shared memory.

Types of Inter-Process Communication Available

IRIX supports several types of IPC. Standard AT&T System V Release 4 IPC is available for making code portable. However, its implementation is fundamentally different from (and slower than) that of the IRIX-specific IPC also provided with IRIX. BSD socket-based IPC is supported for compatibility and, to allow IPC across a network, between processes running on different machines.

Do not mix the various types of IPC in a given program. Use System V IPC—based on a mechanism called keys—for code that must comply with the MIPS ABI, code that needs to be portable, or code that you're porting from another System V operating system. Use arena-based IRIX IPC for applications that require speed, ease of implementation, or multiprocessing

ability. Socket-based IPC is necessary only for code being ported from or to a BSD system, and for network IPC.

This chapter describes the available types of System V and IRIX IPC, and provides examples of each. Since there are many ways to accomplish any given task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency.

System V IPC

This section covers System V IPC and includes:

- “System V IPC Overview”
- “System V Messages”
- “Controlling Message Queues: msgctl()”
- “Operations for Messages: msgsnd() and msgrcv()”
- “System V Semaphores”
- “System V Shared Memory”

System V IPC Overview

System V IPC comprises three inter-process communication mechanisms:

- *Messages* allow processes to send and receive buffers full of data.
- *Semaphores* allow processes to turn on and off a set of flags.
- *Shared memory* gives multiple processes access to the same data area in memory.

System V Messages

The message mechanism allows processes to exchange data stored in buffers. This data is transmitted between processes in discrete units called messages. Processes using this type of IPC can perform two operations: sending messages and receiving messages.

Before a process can send or receive messages, the process must request that the operating system generate a new *message queue* (the mechanism used to control and keep track of messages), and an associated data structure. A process makes this request by using the **msgget()** system call. The requesting process becomes the owner and creator of the resulting message queue, and specifies the initial operation permissions for all processes that might use that queue (including itself). Subsequently, the owning process can relinquish ownership or change the operation permissions using the **msgctl()** system call. However, the creator remains the creator as long as the queue exists. Other processes with permission can use **msgctl()** to perform various other control functions, as described in "Controlling Message Queues: msgctl()."

A process that is attempting to send a message can suspend execution temporarily in order to wait until the process that is to receive the message is ready; similarly, a receiving process can suspend execution until the sending process is ready. Message operations that suspend execution in this fashion are called "blocking message operations." A process that specifies that its execution is not to be suspended—that is, a process that does not wait for communication if such is not immediately available—is performing a "nonblocking message operation."

A blocking message operation can be told to suspend a calling process until one of three conditions occurs:

- The operation is successful.
- The operation receives a signal.
- The message queue is removed.

To request a message operation, the calling process passes arguments to a system call, and the system call attempts to perform its function. If the system call is successful, it returns its results. Otherwise, it returns a known error code (-1), and an external error variable, *errno*, is set accordingly.

Message Queues

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 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

Each message queue has a data structure associated with it. This data structure contains the following information for its message queue:

- operation permissions data
- 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 allowed on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time
- and some padding space reserved for future expansion.

The definition for the message queue structure is:

```
struct msg
{
    struct msg   *msg_next; /* ptr to next msg on queue */
    long         msg_type;  /* message type             */
    short        msg_ts;    /* message text size        */
    caddr_t      msg_spot;  /* message text map address */
};
```

It is located in the */usr/include/sys/msg.h* header file. The definition for the associated data structure, *msqid_ds*, is also located in that header file. The permissions structure that's part of the *msqid_ds* structure is based on another structure called *ipc_perm*, which is also used as a template for permissions for other forms of IPC. The *ipc_perm* data structure format can be found in the */usr/include/sys/ipc.h* header file.

Message Operations Overview

The **msgget()** system call receives an argument *msgflg*, which can be set to indicate various flags. When only the IPC_CREAT flag is set in *msgflg*, **msgget()** performs one of two tasks:

- It gets a new *msqid* and creates an associated message queue and data structure for it; or
- It returns 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*, unless some system-tunable parameter (such as the maximum allowable number of message queues) would be exceeded.

Instead of requesting a specific *key* number, you may indicate a *key* of value zero, which is known as the private *key* (the constant IPC_PRIVATE is defined to be zero). When you specify IPC_PRIVATE for the *key* value, 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, the KEY field for the *msqid* is all zeros, for security reasons.

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, you can specify a control command (IPC_EXCL) in the *msgflg* argument passed to the system call. The details of using this system call are discussed in "Getting Message Queues with msgget()"in this chapter.

When performing the first task, the process that 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 “Controlling Message Queues: msgctl().” 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 and message control can be used.

The available message operations are sending and receiving. System calls are provided for these operations: **msgsnd()** and **msgrcv()**, respectively. Refer to “Operations for Messages: msgsnd() and msgrcv(),” for details of these system calls.

Message control is done by using the **msgctl()** 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 “Controlling Message Queues: msgctl()” for details of the **msgctl()** system call.

Getting Message Queues with msgget()

This section details the **msgget()** system call and provides an example program illustrating its use.

The synopsis in the **msgget(**2**)** reference page looks like this:

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

int msgget (key, msgflg)
key_t key;
int msgflg;
```

All of the listed include-files are located in the */usr/include/sys* directory of the UNIX operating system.

The type of the *key* parameter, *key_t*, is defined by a **typedef** in the *types.h* header file to be equivalent to an integer.

Upon successful completion, **msgget()** returns a message queue identifier. A new *msqid* with an associated message queue and data structure is provided if either of the following is true:

- *key* is equal to IPC_PRIVATE
- *key* is a unique hexadecimal integer, and IPC_CREAT is set in *msgflg*

The value passed to the *msgflg* argument must be an integer type octal value and must specify access permissions, execution modes, and control fields (commands). Access permissions determine the read / write attributes, while execution modes determine the user / group / other attributes of the *msgflg* argument. The permissions and modes are collectively referred to as "operation permissions." Table 1-1 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 1-1 Operation Permissions Codes

Operation Permissions	Octal Value
Read by Owner (MSG_R)	00400
Write by Owner (MSG_W)	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. For instance, if you want a message queue to be readable by its owner and both readable and writable by others, use the code value 00406 (00400 plus 00004 plus 00002). The constants MSG_R and MSG_W, defined in the *msg.h* header file, can be used instead of 00400 and 00200, respectively.

Control commands are constants defined in the *ipc.h* header file. See Table 1-2, which contains the names of the constants that apply to the **msgget()** system call and their values.

Table 1-2 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *msgflg* is therefore a combination of operation permissions and control commands. To accomplish this combination, perform a bitwise OR (|) on the flags with the operation permissions. The bit positions and values for the control commands in relation to those of the operation permissions make this possible.

Two examples:

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

The **msgget()** system call attempts to return a new *msqid* if either of the following conditions is true:

- *key* is equal to IPC_PRIVATE
- *key* does not already have a *msqid* associated with it, and IPC_CREAT is set in *msgflg*

To satisfy the first condition, simply pass IPC_PRIVATE as the *key* argument when calling **msgget()**:

```
msqid = msgget (IPC_PRIVATE, msgflg);
```

The second condition is satisfied if the value for *key* is not already associated with a *msqid* and a bitwise AND of *msgflg* and IPC_CREAT gives “true” (1). This means that the given *key* is not currently being used to refer to any message queue on the computer the program is running on, and that the IPC_CREAT flag is set in *msgflg*.

Note: The system-tunable parameter MSGMNI determines the maximum number of unique message queues (*msqid*s) in the UNIX operating system. Attempting to exceed MSGMNI always causes a failure.

IPC_EXCL is another control flag 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**)** reference page for specific information about the associated data structures. The specific failure conditions, with error names, are listed there as well.

Example Program

The example program in this section (Example 1-1) is a menu-driven program that exercises all possible combinations of the **msgget()** system call.

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**)** reference page. Note that the *errno.h* header file is included instead of declaring *errno* as an external variable; either method works.

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. This choice of names makes the program more readable, and it is perfectly legal since the variables are local to this program.

The variables in 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

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 in the *opperm_flags* variable (lines 36-51).

The system call is made next, and the result is stored in 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()** system call follows in Example 1-1.

Example 1-1 **msgget()** System Call Example

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

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 program*/
```

```
10 main()
11 {
12     key_t key;
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 ");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);
23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags = 0\n");
27     printf("IPC_CREAT = 1\n");
28     printf("IPC_EXCL = 2\n");
29     printf("IPC_CREAT and IPC_EXCL = 3\n");
30     printf(" Flags = ");

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

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

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

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

54     /*Perform the following if the call failed.*/
55     if(msqid == -1)
56     {
57         printf ("\nThe msgget system call failed!\n");
58         printf ("The error number was %d.\n", errno);
59     }
60     /*Return the msqid upon successful completion.*/
61     else
62         printf ("\nThe msqid is %d.\n", msqid);

63     exit(0);
64 }
```

Controlling Message Queues: msgctl()

This section gives a detailed description of using the **msgctl()** system call and an example program that exercises its capabilities.

The synopsis given in the **msgctl(**2**)** reference page 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.

On successful completion, it returns zero; 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 trouble understanding the logic manipulations in this program, read the “Getting Message Queues with msgget()” section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 1-2) is a menu-driven program that exercises all possible combinations of the **msgctl()** system call.

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**)** reference page. Note in this program that *errno* is declared as an external variable, and therefore, the *errno.h* header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is

perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

uid	used to store the IPC_SET value for the effective user identification
gid	used to store the IPC_SET value for the effective group identification
mode	used to store the IPC_SET value for the operation permissions
bytes	used to store the IPC_SET value for the number of bytes in the message queue (*msg_qbytes*)
rtrn	used to store the return integer value from the system call
msqid	used to store and pass the message queue identifier to the system call
command	used to store the code for the desired control command so that subsequent processing can be performed on it
choice	used to determine which member is to be changed for the IPC_SET control command
msqid_ds	used to receive the specified message queue identifier's data structure when an IPC_STAT control command is performed
**buf*	a pointer passed to the system call; it 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 in 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 in 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 in 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 in 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 in Example 1-2.

Example 1-2 **msgctl()** System Call Example

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

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

10  /*Start of main C language program*/
11  main()
12  {
13    extern int errno;
14    int uid, gid, mode, bytes;
15    int rtrn, msqid, command, choice;
16    struct msqid_ds msqid_ds, *buf;
17    buf = &msqid_ds;
18    /*Get the msqid, and command.*/
19    printf("Enter the msqid = ");
20    scanf("%d", &msqid);
21    printf("\nEnter the number for\n");
22    printf("the desired command:\n");
23    printf("IPC_STAT = 1\n"); 24 printf("IPC_SET = 2\n");
25    printf("IPC_RMID = 3\n");
26    printf("Entry = ");
27    scanf("%d", &command);
28    /*Check the values.*/
29   printf (\nmsqid =%d, command = %\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      rtrn = msgctl(msqid, IPC_STAT,
38       buf);
39      printf ("\nThe USER ID = %d\n",
40        buf->msg_perm.uid);
41      printf ("The GROUP ID = %d\n",
42        buf->msg_perm.gid);
43      printf ("The operation permissions = 0%o\n",
```

```
44        buf->msg_perm.mode);
45      printf ("The msg_qbytes = %d\n",
46        buf->msg_qbytes);
47      break;
48     case 2: /*Select and change the desired
49           member(s) of the data structure.*/
50        /*Get the original data for this msqid
51           data structure first.*/
52        rtrn = 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 = ");
60        scanf("%d", &choice);
61        /*Only one choice is allowed per pass
62           as an illegal entry will
63           cause repetitive failures until
64           msqid_ds is updated with
65           IPC_STAT.*/
66        switch(choice){
67        case 1:
68          printf("\nEnter USER ID = ");
69          scanf ("%d", &uid);
70          buf->msg_perm.uid = uid;
71          printf("\nUSER ID = %d\n",
72            buf->msg_perm.uid);
73          break;
74        case 2:
75          printf("\nEnter GROUP ID = ");
76          scanf("%d", &gid);
77          buf->msg_perm.gid = gid;
78          printf("\nGROUP ID = %d\n",
79            buf->msg_perm.gid);
80          break;
81        case 3:
82          printf("\nEnter MODE = ");
83          scanf("%o", &mode);
84          buf->msg_perm.mode = mode;
85          printf("\nMODE = 0%o\n",
86            buf->msg_perm.mode);
87          break;
88        case 4:
```

```
89          printf("\nEnter 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        rtrn = 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       rtrn = msgctl(msqid, IPC_RMID, NULL);
104     }
105     /*Perform the following if call is  unsuccessful.*/
106     if(rtrn == -1)
107     {
108       printf ("\nThe msgctl system call failed!\n");
109       printf ("The error number = %d\n", errno);
110     }
111     /*Return the msqid upon successful completion.*/
112     else
113       printf ("\nMsgctl was successful for msqid = %d\n",
114         msqid);
115     exit (0);
116  }
```

Operations for Messages: msgsnd() and msgrcv()

This section describes the **msgsnd()** and **msgrcv()** system calls and presents an example program that exercises all of their capabilities.

The synopsis found in the **msgop(**2**)** reference page, which describes both **msgsnd()** and **msgrcv()**, is as follows:

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

int msgsnd(int msqid, const void *msgp,
 size_t msgsz, int msgflg);
```

```
int msgrcv(int msqid, void *msgp,
 size_t 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; 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 occurs 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 fails and returns -1.

Further details of this system call are discussed in the example program for it. If you don’t understand the logic manipulations in this program, read the “Getting Message Queues with msgget()” section of this chapter; it goes into more detail than 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 a -1 is returned.

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 receives 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 fails immediately when a message of the desired type is not on the queue. *msgflg* can also specify that the system call fails 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 don't understand the logic manipulations in this program, read the "Getting Message Queues with msgget()" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 1-3) is a menu-driven program that exercises all possible combinations of the **msgsnd()** and **msgrcv()** system calls.

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)** reference page. Note that in this program, *errno* is declared as an external variable. 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 shown in Table 1-3:

Table 1-3 Variables Used in the **msgop()** Example Program

Variable	Purpose
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 workstation where in *msgbuf* it is set to one 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.

Table 1-3 (continued) Variables Used in the **msgop()** Example Program

Variable	Purpose
i	Used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd()** system call; it is also used as a counter to output the received message for the **msgrcv()** system call.
c	Used to receive the input character from the **getchar()** function (line 50).
flag	Used to store the code of IPC_NOWAIT for the **msgsnd()** system call (line 61).
flags	Used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv()** system call (line 117).
choice	Used to store the code for sending or receiving (line 30).
rtrn	Used to store the return values from all system calls.
msqid	Used to store and pass the desired message queue identifier for both system calls.
msgsz	Used to store and pass the size of the message to be sent or received.
msgflg	Used to pass the value of flag for sending or the value of flags for receiving.
msgtyp	Used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a *msqid_ds* data structure is set up in the program (line 21) with a pointer that is initialized to point to it (line 22); this allows the data structure members that are affected by message operations to be observed. They are observed by using the **msgctl()** system call (with IPC_STAT) to get them so the program can 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 in 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 in 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 continues until an end of file is reached (for the **getchar()** function, this is a control-D immediately following a carriage return). 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; therefore, the message appears 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 1 be entered for yes or anything else for no (lines 57-65). The result is stored in 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 then 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 (which should be zero) is printed (lines 73-76).

Every time a message is successfully sent, three members of the associated data structure are updated. They are:

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 in *msqid* (lines 100-103).

The message type is requested, and it is stored in *msgtyp* (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored in *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 in *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 success, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, three members of the associated data structure 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 is shown in Example 1-3.

Example 1-3 **msgsnd()** and **msgrcv()** System Call Example

```
1  /*This is a program to illustrate
2   * the message operations, msgop(),
3   * system call capabilities: msgsnd() and msgrcv().
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;

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 or\n”);
35      printf(“the message queue to\n”);
36      printf(“handle the message = “);
37      scanf(“%d”, &msqid);

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

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

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

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

52      /*Determine the message size.*/
53      msgsz = i + 1;
54      /*Echo the message to send.*/
55      for(i = 0; i < msgsz; i++)
56        putchar(sndbuf.mtext[i]);

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

66      /*Check the msgflg.*/
67      printf(“\nmsgflg = 0%o\n”, msgflg);

68      /*Send the message.*/
69      rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
```

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

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

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

82          /*Print out the affected members.*/

83          /*Print the incremented number of
84            messages on the queue.*/
85          printf("\nThe msg_qnum = %d\n",
86            buf->msg_qnum);
87          /*Print the process id of the last sender.*/
88          printf("The msg_lspid = %d\n",
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);

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 codeen");
111    printf("to select the desired flags: \n");
112    printf("No flags = 0\n");
113    printf("MSG_NOERROR = 1\n");
114    printf("IPC_NOWAIT = 2\n");
115    printf("MSG_NOERROR and IPC_NOWAIT = 3\n");
116    printf(" Flags = ");
117    scanf("%d", &flags);

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

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

138    /*Check the values for the arguments.*/
139    printf("\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);

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 }
```

System V Semaphores

Semaphore IPC allows processes to communicate through the exchange of data items called semaphores. A single semaphore is represented as a positive integer (0 through 32,767). Semaphores are usually used to manage resources; each semaphore indicates whether or not a specific data item is currently in use (and by how many different processes). 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 at one time. A semaphore set can contain one or more semaphores, up to a limit set by the system administrator. (This limit, a tunable parameter called SEMMSL, has a default value of 25.) To create a set of semaphores, use the **semget()** system call.

The process performing the **semget()** system call becomes the owner / creator of the semaphore set, determines how many semaphores are in the set, and sets the operation permissions for the set. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl()** (semaphore control) system call.The creating process remains the creator as long as the semaphore set exists, but other processes with permission can use **semctl()** to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore set. Each semaphore within a set can be either increased or decreased with the **semop(**2**)** system call (see the IRIX reference pages for more information).

To increase a semaphore, pass a positive integer value of the desired magnitude to the **semop()** system call. To decrease a semaphore, pass a negative integer value of the desired magnitude.

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 that 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()** 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()**, semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore in the set is numbered one less than the number of semaphores 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 when all operations can be performed successfully, that the operation should be undone. Otherwise, the operations are performed and the semaphores are changed, or one “nonblocking operation” is unsuccessful and none are changed. 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 that 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 attempts to perform 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 you can use semaphores, you must request a uniquely identified semaphore set and its associated data structure with a system call. The unique identifier is called the semaphore identifier (*semid*); it is used to reference a particular semaphore set and data structure.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The user may select the number of semaphores (*nsems*) in a set. The following members are in each instance of the structure within a semaphore set:

- semaphore text map address
- process identification (PID) performing last operation
- number of processes waiting for the semaphore value to become greater than its current value
- number of processes waiting for the semaphore value to equal zero

Each semaphore set has an associated data structure. This data structure contains information related to the semaphore set:

- operation permissions data
- pointer to first semaphore in the set
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The definition for the semaphore set is:

```
struct sem
{
    ushort semval; /* semaphore text map address */
    short sempid; /* pid of last operation */
    ushort semncnt; /* # awaiting semval > cval */
    ushort semzcnt; /* # awaiting semval = 0 */
};
```

This definition is located in the */usr/include/sys/sem.h* header file. The structure definition for the associated semaphore data structure is also located in the same header file. Note that the *sem_perm* member of this structure uses *ipc_perm* as a template. The *ipc_perm* data structure is the same for all IPC facilities, and it is located in the */usr/include/sys/ipc.h* header file.

The **semget()** 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()** 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.

Also, a provision exists for specifying a *key* of value zero, 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 that 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 “Controlling Semaphores with semctl()” for more information. 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 you do not desire to have an existing *semid* returned, a control command (IPC_EXCL) can be specified in the *semflg* argument passed to the system call. The system call fails 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 the **semget()** system call are discussed in "Getting Semaphores with semget()."

Once a uniquely-identified semaphore set and data structure are created, use semaphore operations (**semop()**) and semaphore control (**semctl()**).

Semaphores can be increased, decreased, or tested for zero. A single system call, **semop()**, is used to perform these operations. Refer to "Operations on Semaphores: semop()" for details on this system call.

Semaphore control is done by using the **semctl()** system call. Use control operations 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 and 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 “Controlling Semaphores with semctl()” for details of the **semctl()** system call.

Getting Semaphores with semget()

This section contains a detailed description of using the **semget()** system call along with an example program illustrating its use.

The synopsis found in the **semget(**2**)** reference page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

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 must 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.”

Table 1-4 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 1-4 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

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 that can be used for the owner. They are:

```
SEM_A 0200 /* alter permission by owner */
SEM_R 0400 /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Table 1-5 contains the names of the constants that apply to the **semget()** system call along with their values. They are also referred to as flags and are defined in the *ipc.h* header file.

Table 1-5 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

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 using a bitwise OR (|) with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

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**)** reference page, success or failure of this system call depends upon the actual argument values for *key*, *nsems*, *semflg* or system tunable parameters. The system call attempts 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”

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);
```

which causes the system call to be attempted because it satisfies the first condition specified.

Exceeding the SEMMNI, SEMMNS, or SEMMSL system-tunable parameters always causes 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 already in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag

has been set (using *semflg* | IPC_CREAT). 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**)** reference page for specific associated data structure initialization for successful completion.

Example Program

The example program in this section (Example 1-4) is a menu-driven program that exercises all possible combinations of the **semget()** system call.

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**)** reference page. Note that the *errno.h* header file is included as opposed to declaring *errno* as an external variable; either method works.

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 you to observe the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in 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 in *nsems*.

The system call is made next, and the result is stored in 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()** system call is shown in Example 1-4.

Example 1-4 **semget()** System Call Example

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

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

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

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

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

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

33     /*Error checking (debugging)*/
34     printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35       key, opperm, flags);
```

```
36     /*Incorporate the control fields (flags) with
37       the operation permissions.*/
38     switch (flags)
39     {
40     case 0: /*No flags are to be set.*/
41       opperm_flags = (opperm | 0);
42       break;
43     case 1: /*Set the IPC_CREAT flag.*/
44       opperm_flags = (opperm | IPC_CREAT);
45       break;
46     case 2: /*Set the IPC_EXCL flag.*/
47       opperm_flags = (opperm | IPC_EXCL);
48       break;
49     case 3: /*Set the IPC_CREAT and IPC_EXCL
50         flags.*/
51       opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52     }
53     /*Get the number of semaphores for this set.*/
54     printf("\nEnter the number of\n");
55     printf("desired semaphores for\n");
56     printf("this set (25 max) = ");
57     scanf("%d", &nsems);
58     /*Check the entry.*/
59     printf("enNsems = %d\n", nsems);
60     /*Call the semget system call.*/
61     semid = semget(key, nsems, opperm_flags);
62     /*Perform the following if the call is unsuccessful.*/
63     if(semid == -1)
64     {
65       printf("The semget system call failed!\n");
66       printf("The error number = %d\n", errno);
67     }
68     /*Return the semid upon successful completion.*/
69     else
70       printf("\nThe semid = %d\n", semid);
71     exit(0);
72   }
```

Controlling Semaphores with semctl()

This section details the **semctl()** system call and provides an example program that exercises all of its capabilities.

The synopsis found in the **semctl(**2**)** reference page is as follows:

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

int semctl (semid, semnum, cmd, arg)
int semid, semnum, cmd;
union semun

{
      int val;
      struct semid_ds *bu;
      ushort array[];
} arg;
```

The **semctl()** 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()** 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.

Use the *arg* argument to pass the appropriate union member to the system call 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 “Getting Semaphores with semget()”; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-10) is a menu-driven program that exercises all possible combinations of using the **semctl()** system call.

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**)** reference page. 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()** 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 ID 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. It 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 all of the required declarations have been presented for this program. The next paragraphs explain how it works.

First, the program prompts for a valid semaphore set identifier, which is stored in the *semid* variable (lines 25-27). This is required for all **semctl()** system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored in 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 in 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 in 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 in 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 in 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 that 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 that 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()** 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 in 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 in the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending on 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* is removed from the UNIX operating system along with its associated data structure and semaphore set.

Depending on success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl()** system call is shown in Example 1-5.

Example 1-5 **semctl()** System Call Example

```
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;

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      {

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;
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;
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 = %den", 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.*/
125       /*Get and print the current status values.*/
127       retrn = semctl(semid, 0, IPC_STAT, arg.buf);
128       printf (“\nThe USER ID = %d\n”,
129         arg.buf->sem_perm.uid);
130       printf (“The GROUP ID = %d\n”,
131         arg.buf->sem_perm.gid);
132       printf (“The operation permissions = 0%o\n”,
133         arg.buf->sem_perm.mode);
134       printf (“The number of semaphores in set = %d\n”,
135         arg.buf->sem_nsems);
136       printf (“The last semop time = %d\n”,
137         arg.buf->sem_otime);
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       retrn = semctl(semid, 0, IPC_STAT, arg.buf);
145       if(retrn == -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(“enUSER ID = %d\n”,
161           arg.buf->sem_perm.uid);
162         break;
163       case 2: /*Change the group ID.*/
164         printf(“\nEnter GROUP ID = “);
165         scanf(“%d”, &gid);
166           arg.buf->sem_perm.gid = gid;
167         printf(“\nGROUP ID = %d\n”,
168           arg.buf->sem_perm.gid);
```

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

Operations on Semaphores: semop()

This section details the **semop()** system call and provides an example program that exercises its capabilities.

The synopsis found in the **semop(**2**)** reference page 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;
size_t nsops;
```

semop() returns zero on success, or -1 on failure.

The *semid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **semget()** 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 */usr/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()** 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 returns unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call is 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 (Example 1-6) is a menu-driven program that exercises all possible combinations of the **semop()** system call.

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 **semop(**2**)** reference page. 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()** 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()** 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 in 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 in 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 **semctl()** with the GETALL control command.

The example program for the **semop()** system call is shown in Example 1-6.

Example 1-6 **semop()** System Call Example

```
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("ensemid = %d", semid);
25    /*Enter the number of operations.*/
26    printf("\nEnter the number of semaphore\n");
27    printf("operations for this set = ");
28    scanf("%d", &nsops);
29    printf("ennosops = %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("ensem_op = %d\n", sops->sem_op);
51     /*Specify the desired flags.*/
52     printf("\nEnter the corresponding\n");
53     printf("number for the desired\n");
54     printf("flags:\n");
55     printf("No flags = 0\n");
56     printf("IPC_NOWAIT = 1\n");
57     printf("SEM_UNDO = 2\n");
58     printf("IPC_NOWAIT and SEM_UNDO = 3\n");
59     printf(" Flags = ");
60     scanf("%d", &flags);

61     switch(flags)
62     {
63     case 0:
64       sops->sem_flg = 0;
65       break;
66     case 1:
67       sops->sem_flg = IPC_NOWAIT;
68       break;
69     case 2:
70       sops->sem_flg = SEM_UNDO;
71       break;
72     case 3:
73       sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74       break;
```

```
75      }
76      printf("\nFlags = 0%o\n", sops->sem_flg);
77    }
78    /*Print out each structure in the array.*/
79    for(i = 0; i < nsops; i++)
80    {
81        printf("\nsem_num = %d\n", sembuf[i].sem_num);
82        printf("sem_op = %d\n", sembuf[i].sem_op);
83        printf("sem_flg = %o\n", sembuf[i].sem_flg);
84        printf("%c", ' ');
85      }

86      sops = sembuf; /*Reset the pointer to
87          sembuf[0].*/
88      /*Do the semop system call.*/
89      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  }
```

System V Shared Memory

Shared memory IPC allows two or more executing processes 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 depends on memory management hardware.

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()** 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()** - shared memory attach
- **shmdt()** - 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()** 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()** system call.

System calls, documented in the IRIX reference pages, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call attempts to perform 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 memory sharing 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 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 *ipc_perm* data structure is the same for all IPC facilities, and is located in the */usr/include/sys/ipc.h* header file.

The **shmget()** 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()** 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.

A provision exists for specifying a *key* of value zero, 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 you do not want to have an existing *shmid* returned, a control command (IPC_EXCL) can be set in the *shmflg* argument passed to the system call. The details of using this system call are discussed in “Getting Shared Memory Segments with shmget().”

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 “Controlling Shared Memory: shmctl().” 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 and control 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()** and **shmdt()**. Refer to “Operations for Shared Memory: shmat() and shmdt()” for details of these system calls.

Shared memory segment control is done by using the **shmctl()** 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 "Controlling Shared Memory: shmctl()" for details of the **shmctl()** system call.

Getting Shared Memory Segments with shmget()

This section gives a detailed description of using the **shmget()** system call along with an example program illustrating its use.

The synopsis found in the **shmget(**2**)** reference page 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 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()** 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 of the following is true:

- *key* is equal to IPC_PRIVATE
- *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 must 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." Table 1-6 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 1-6 Operation Permissions Codes

Operation Permissions	Octal Values
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

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 are desired, the code value would be 00406 (00400 plus 00006).

Constants located in the *shm.h* header file can be used for the owner. They include:

```
SHM_R 0400
SHM_W 0200
```

Control commands are predefined constants (represented by all uppercase letters). Table 1-7 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.

Table 1-7 Control Commands (Flags)

Control	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

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 using bitwise OR (|) with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

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

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

As specified by the **shmget(2)** reference page, success or failure of this system call depends upon the argument values for *key*, *size*, and *shmflg* or system tunable parameters. The system call attempts 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);
```

which causes the system call to be attempted because it satisfies the first condition specified. Exceeding the SHMMNI system-tunable parameter always causes a failure. The SHMMNI system-tunable parameter determines the maximum number of unique shared memory segments (*shmids*) in the UNIX operating system.

The second condition is satisfied if the value for *key* is not already associated with a *shmid* and the bitwise ANDing of *shmflg* and IPC_CREAT is “true” (not zero). This means that the *key* is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag has been set (by using *shmflg* | IPC_CREAT). 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 fails 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**)** reference 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 (Example 1-7) is a menu-driven program that exercises all possible combinations of the **shmget()** system call.

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

This program begins (lines 4-7) by including the required header files as specified by the **shmget(**2**)** reference page. Note that the *errno.h* header file is included as opposed to declaring *errno* as an external variable; either method works.

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. Thus you can observe the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the *opperm_flags* variable (lines 35-50).

A display then prompts for the size of the shared memory segment, which is stored in the *size* variable (lines 51-54).

The system call is made next, and the result is stored in 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()** system call is shown in Example 1-7.

Example 1-7 shmget() System Call Example

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

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

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

17     /*Enter the desired octal operation
18       permissions.*/
19     printf("\nEnter the operation\n");
20     printf("permissions in octal = ");
21     scanf("%o", &opperm);
22     /*Set the desired flags.*/
23     printf("\nEnter corresponding number to\n");
24     printf("set the desired flags:\n");
25     printf("No flags = 0\n");
26     printf("IPC_CREAT = 1\n");
27     printf("IPC_EXCL = 2\n");
28     printf("IPC_CREAT and IPC_EXCL = 3\n");
29     printf(" Flags = ");
```

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

32     /*Check the values.*/
33     printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34       key, opperm, flags);
35     /*Incorporate the control fields (flags) with
36       the operation permissions*/
37     switch (flags)
38     {
39     case 0: /*No flags are to be set.*/
40       opperm_flags = (opperm | 0);
41       break;
42     case 1: /*Set the IPC_CREAT flag.*/
43       opperm_flags = (opperm | IPC_CREAT);
44       break;
45     case 2: /*Set the IPC_EXCL flag.*/
46       opperm_flags = (opperm | IPC_EXCL);
47       break;
48     case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
49       opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50     }
51     /*Get the size of the segment in bytes.*/
52     printf ("\nEnter the segment");
53     printf ("\nsize in bytes = ");
54     scanf ("%d", &size);

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

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

Controlling Shared Memory: shmctl()

This section details using the **shmctl()** system call and provides an example program that exercises all of its capabilities.

The synopsis found in the **shmctl(**2**)** reference page is as follows:

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

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

The **shmctl()** system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned; when unsuccessful, **shmctl()** returns a -1.

The *shmid* variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget()** system call.

The *cmd* argument can be replaced by one of following control commands (flags):

- IPC_STAT: return the status information contained in the associated data structure for the specified *shmid* and place it in the data structure pointed to by the **buf* pointer in the user memory area
- IPC_SET: for the specified *shmid*, set the effective user and group identification, and operation permissions
- IPC_RMID: remove the specified *shmid* along with its associated shared memory segment data structure
- SHM_LOCK: lock the specified shared memory segment in memory; must be super user
- SHM_UNLOCK: unlock the shared memory segment from memory; must be super user.

A process must have an effective user identification of OWNER/CREATOR or super user to perform an IPC_SET or IPC_RMID control command. Only

the super user can perform a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read “Getting Shared Memory Segments with shmget()”; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 1-8) is a menu-driven program that exercises all possible combinations of the **shmctl()** system call.

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**)** reference page. 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 that is stored in the *shmid* variable (lines 18-20). This is required for every **shmctl()** system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored in 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 in 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 in 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()** system call is shown in Example 1-8.

Example 1-8 shmctl() System Call Example

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

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

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

18     /*Get the shmid, and command.*/
19     printf("Enter the shmid = ");
20     scanf("%d", &shmid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");
23     printf("IPC_STAT = 1\n");
24     printf("IPC_SET = 2\n");
25     printf("IPC_RMID = 3\n");
26     printf("SHM_LOCK = 4\n");
27     printf("SHM_UNLOCK = 5\n");
28     printf("Entry = ");
29     scanf("%d", &command);

30     /*Check the values.*/
31     printf ("\nshmid =%d, command = %d\n",
32     shmid, command);
33     switch (command)
34     {
35     case 1: /*Use shmctl() to duplicate
36       the data structure for
37         shmid in the shmid_ds area pointed
```

```
38          to by buf and then print it out.*/
39        rtrn = shmctl(shmid, IPC_STAT,
40          buf);
41        printf (“\nThe USER ID = %d\n”,
42          buf->shm_perm.uid);
43        printf (“The GROUP ID = %d\n”,
44          buf->shm_perm.gid);
45        printf (“The creator’s ID = %d\n”,
46          buf->shm_perm.cuid);
47        printf (“The creator’s group ID = %d\n”,
48          buf->shm_perm.cgid);
49        printf (“The operation permissions = 0%o\n”,
50          buf->shm_perm.mode);
51        printf (“The slot usage sequenceen”);
52        printf (“number = 0%x\n”,
53          buf->shm_perm.seq);
54        printf (“The key= 0%x\n”,
55          buf->shm_perm.key);
56        printf (“The segment size = %d\n”,
57          buf->shm_segsz);
58        printf (“The pid of last shmop = %d\n”,
59          buf->shm_lpid);
60        printf (“The pid of creator = %d\n”,
61          buf->shm_cpid);
62        printf (“The current # attached = %d\n”,
63          buf->shm_nattch);
64        printf(“The in memory # attached = %d\n”,
65          buf->shm_cnattach);
66        printf(“The last shmat time = %d\n”,
67          buf->shm_atime);
68        printf(“The last shmdt time = %d\n”,
69          buf->shm_dtime);
70        printf(“The last change time = %d\n”,
71          buf->shm_ctime);
72        break;
          /* Lines 73 - 87 deleted */
88      case 2: /*Select and change the desired
89        member(s) of the data structure.*/
90        /*Get the original data for this shmid
91           data structure first.*/
92        rtrn = shmctl(shmid, IPC_STAT, buf);
93        printf(“\nEnter the number for the\n”);
94        printf(“member to be changed:\n”);
95        printf(“shm_perm.uid = 1\n”);
96        printf(“shm_perm.gid = 2\n”);
```

```
97        printf(“shm_perm.mode = 3\n”);
98        printf(“Entry = “);
99        scanf(“%d”, &choice);
100       /*Only one choice is allowed per
101         pass as an illegal entry will
102         cause repetitive failures until
103          shmid_ds is updated with
104          IPC_STAT.*/
105       switch(choice){
106       case 1:
107         printf(“\nEnter USER ID = “);
108         scanf (“%d”, &uid);
109         buf->shm_perm.uid = uid;
110         printf(“\nUSER ID = %d\n”,
111         buf->shm_perm.uid);
112         break;
113       case 2:
114         printf(“\nEnter GROUP ID = “);
115         scanf(“%d”, &gid);
116         buf->shm_perm.gid = gid;
117         printf(“\nGROUP ID = %d\n”,
118         buf->shm_perm.gid);
119         break;

120       case 3:
121         printf(“\nEnter MODE = “);
122         scanf(“%o”, &mode);
123         buf->shm_perm.mode = mode;
124         printf(“\nMODE = 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;
132     case 3: /*Remove the shmid along with its
133       associated
134       data structure.*/
135       rtrn = shmctl(shmid, IPC_RMID, NULL);
136       break;
137     case 4: /*Lock the shared memory segment*/
138       rtrn = shmctl(shmid, SHM_LOCK, NULL);
139       break;
140     case 5: /*Unlock the shared memory
```

```
141      segment.*/
142      rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
143      break;
144    }
145    /*Perform the following if the call fails.*/
146    if(rtrn == -1)
147    {
148      printf ("\nThe shmctl system call failed!\n");
149      printf ("The error number = %d\n", errno);
150    }
151    /*Return the shmid upon successful completion.*/
152    else
153      printf ("\nShmctl was successful for shmid = %d\n",
154      shmid);
155    exit (0);
156 }
```

Operations for Shared Memory: shmat() and shmdt()

This section details the **shmat()** and **shmdt()** system calls, and presents an example program that exercises all of their capabilities.

The synopsis found in the **shmop(**2**)** reference page, which includes both **shmat()** and **shmaddr()**, 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 to a Shared Memory Segment

The **shmat()** 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 is the address in core memory where the process is attached to the shared memory segment, and when unsuccessful, it is -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()** system call.

The *shmaddr* argument can be zero or user-supplied when passed to the **shmat()** 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. Here are some typical address ranges:

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

Note that these addresses are in chunks of 20,000 hexadecimal. It would be wise to let the operating system pick addresses so as to improve portability.

The *shmflg* argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat()** system call.

Further details are discussed in the example program for **shmop()**. If you have problems understanding the logic manipulations in this program, read “Getting Shared Memory Segments with shmget()”; it goes into more detail than would be practical to do for every system call.

Detaching Shared Memory Segments

The **shmdt()** system call requires one argument to be passed to it, and it returns an integer value.

shmdt() returns zero if it completes successfully; otherwise, it returns -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 “Getting Shared Memory Segments with shmget()”; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8-17) is a menu-driven program that exercises all possible combinations of the **shmat()** and **shmdt()** system calls.

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

This program begins (lines 5-9) by including the required header files as specified by the **shmop(**2**)** reference page. 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 were chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. The names make the program more readable; this is legal since they are local to the program. The variables declared for this program and their purposes include:

- *flags*: used to store the codes of SHM_RND or SHM_RDONLY for the **shmat()** system call
- *addr*: used to store the address of the shared memory segment for the **shmat()** and **shmdt()** 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()** system call
- *rtrn*: 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()** and **shmdt()** 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 in 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 in 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 in 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 in 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()** 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 in 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 in the *addr* variable (line 84). Then, the **shmdt()** 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()** system calls is shown in Example 1-9.

Example 1-9 **shmop()** System Call Example

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

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

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

21     scanf("%d", &attach);
22     printf("Number of attaches = %d\n", attach);
23     for(i = 1; i <= attach; i++) {
24       /*Enter the shared memory ID.*/
25       printf("\nEnter the shmid ofen");
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 desireden”);
40        printf(“flags:\n”);
41        printf(“SHM_RND = 1\n”);
42        printf(“SHM_RDONLY = 2\n”);
43        printf(“SHM_RND and SHM_RDONLY = 3\n”);
44        printf(“ Flags = “);
45        scanf(“%d”, &flags);

46        switch(flags)
47        {
48        case 1:
49          shmflg = SHM_RND;
50          break;
51        case 2:
52          shmflg = SHM_RDONLY;
53          break;
54        case 3:
55          shmflg = SHM_RND | SHM_RDONLY;
56          break;
57        }
58        printf(“\nFlags = 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 ofen”);
73    printf(“detachments for this\n”);
74    printf(“process (1-4).\n”);
75    printf(“ Detachments = “);
76    scanf(“%d”, &detach);
77    printf(“Number of attaches = %d\n”, detach);
78    for(i = 1; i <= detach; i++) {

79      /*Enter the value for shmaddr.*/
80      printf(“\nEnter the value for\n”);
```

```
81      printf(“the shared memory address\n”);
82      printf(“in hexadecimal:\n”);
83      printf(“ Shmaddr = “);
84      scanf(“%x”, &addr);
85      printf(“The desired address = 0x%x\n”, addr);

86      /*Do the shmdt system call.*/
87      retrn = (int)shmdt(addr);
88      if(retrn == -1) {
89        printf(“Error = %d\n”, errno);
90      }
91      else {
92        printf (“\nShmdt was successful\n”);
93        printf(“for address = 0%x\n”, addr);
94      }
95    }
96 }
```

IRIX IPC

To meet the demands of parallel programming, IRIX provides a set of fast, low-overhead inter-process communication mechanisms. These mechanisms are powerful and easy to use. However, remember that they are IRIX-specific, so code using them is not portable to other systems.

Unlike System V IPC mechanisms that use their own namespace, IRIX IPC mechanisms are associated with the filesystem namespace. To begin using IRIX IPC, a process must specify the name of a file to be used as a *shared arena*. All processes using the same arena have access to the same set of IPC mechanisms. This makes it relatively easy for unrelated processes to communicate using IRIX IPC. The shared arena file is mapped into the process’s user space, which means that most of the shared arena IPC functions do not have to make system calls. This is one reason that the overhead on IRIX IPC is lower than the overhead on standard System V IPC.

IRIX IPC comprises four main mechanisms, two of which—semaphores and shared memory—resemble their System V counterparts. The other mechanisms are *spinlocks*, simple busy-wait locks for low-level synchronization, and *barriers*, which provide rendezvous points for multiple processes.

Note: Modules using IRIX IPC routines should include <*stdio.h*> and <*ulocks.h*>, and should be linked with the *libmpc.so* shared object (**-lmpc**).

For more information on any of the routines presented here, see the appropriate reference pages.

This section explains how to use IRIX IPC. Topics include;

- "Initializing the Shared Arena" describes how to begin using shared arenas.
- "Using Shared-Arena Semaphores" explains how to allocate and change the value of a shared-arena semaphore.
- "Using Spinlocks" covers how to allocate, lock, and unlock spinlocks.
- "Using IRIX Shared Memory" describes how to allocate and free shared memory.
- "Using Barriers" covers how to allocate, use, and free a barrier.
- "Exchanging the First Datum" explains how to communicate the location of an object in an arena to another process using the arena.

Initializing the Shared Arena

To begin using IRIX IPC, call **usinit()**.

Syntax

```
#include <stdio.h>
#include <ulocks.h>
usptr_t *usinit (const char *filename);
```

The *filename* variable specifies a file to use as the shared arena. If the file doesn't exist, **usinit()** creates it. If a shared arena exists by that name, **usinit()** joins the shared arena. If the file exists but isn't a shared arena, **usinit()** overwrites it. In any case, **usinit()** is subject to regular filesystem permissions: it returns an error if the process doesn't have read and write permission on the file (if it already exists) or permission to create the file (if it doesn't exist).

Using Shared-Arena Semaphores

To allocate a new shared-arena semaphore, call **usnewsema()**.

Syntax

```
#include <ulocks.h>
usptr_t *usnewsema (usptr_t *handle, int initial_value);
```

The *initial_value* argument simply specifies the initial value of the semaphore. An initial value of zero is commonly used for a synchronization semaphore: the first process that attempts a P operation on the semaphore blocks until another process performs a V operation on the semaphore. An initial value of 1 may be used to provide a simple mutual exclusion semaphore: the first process attempting a P operation on the semaphore succeeds, and subsequent processes block until the first process releases the semaphore by performing a V operation on it.

Changing the Values of Shared-Arena Semaphores

To perform P and V operations on shared arena semaphores, use the **uspsema()** and **usvsema()** functions. The **uscpsema()** function provides a conditional P operation: it performs a P operation on the semaphore only if it can do so without blocking. The **ustestsema()** function returns the current value of the semaphore (useful primarily for debugging), and the **usinitsema()** function reinitializes the semaphore to a specified value. Note that if you reinitialize a semaphore on which processes are waiting, the processes will not be woken.

Syntax

```
#include <ulocks.h>
int uspsema (usema_t *sema);
int uscpsema (usema_t *sema);
int usvsema (usema_t *sema);
int ustestsema (usema_t *sema);
int usinitsema (usema_t *sema, int initial_value);
```

The following code fragment demonstrates the use of **uspsema()**, **usvsema()**, and **uscpsema()**.

Example 1-10 Using **uspsema()**, **usvsema()**, and **uscpsema()**

```
char *arenafile = "/usr/tmp/testarena";
usptr_t arena;
usema_t *sema;
/* open an arena, allocate a semaphore */
arena = usinit(arenafile);
if (arena == NULL)
{
  /* error */
}
sema = usnewsema(arena);
if (sema == NULL)
{
  /* error */
}
/* acquire the semaphore */
uspsema(sema);
/* release the semaphore */
usvsema(sema);
/* try to get the semaphore again, without blocking */
if (uscpsema(sema) == 1)
{
 /*we succeeded, so we have to release the semaphore again*/
  usvsema(sema);
}
else
  /* failed to get the semaphore */
```

Using Spinlocks

Spinlocks are somewhat like semaphores. Instead of having an integer value, a spinlock has a binary value: it is either locked or unlocked.

The way that spinlocks are implemented depends upon the hardware architecture of the computer using them. On multiprocessor computers, spinlocks are busy-wait locks, so the processor continually tries to acquire the lock until it succeeds. This implementation only makes sense on multiprocessor systems, in which one processor can release the lock while another processor is "spinning" trying to acquire the lock. On single processor machines, spinlocks are implemented using the same algorithm as semaphores—that is, processes waiting to acquire a lock may be put to sleep until the lock is released by another process.

To allocate a new spinlock, use **usnewlock**():

Syntax

```
#include <ulocks.h>
ulock_t usnewlock (usptr_t *arena);
```

The **usnewlock**() function returns a lock, which may then be used by the process. Locking and unlocking can be performed with the functions **ussetlock**(), **uscsetlock**(), **uswsetlock**(), and **usunsetlock**().

Syntax

```
#include <ulocks.h>
int ussetlock (ulock_t lock);
int uscsetlock (ulock_t lock, unsigned spins);
int uswsetlock (ulock_t lock, unsigned spins);
int usunsetlock (ulock_t lock);
```

The **ussetlock()** function locks the specified lock. On multiprocessor systems it spins until it succeeds; on single-processor systems, it may sleep until it can set the lock. It returns 1, unless it encounters an error. The **uscsetlock()** function tries to set the lock *spins* times; if it succeeds, it returns 1, if it fails, it returns 0. On a single-processor system, uscsetlock() ignores the spins argument and only sets the lock if it can do so without waiting. The **uswsetlock()** function resembles **ussetlock()** except that after every *spins* attempts to set the lock, it yields the processor by calling **sginap()**. On single-processor systems, **uswsetlock()** behaves exactly like **ussetlock()**. The **usunsetlock()** function unlocks the lock; it always returns 0, unless it encounters an error. All of these functions return -1 in the event of an error. You can find the reason for the error by calling **oserror()**.

IRIX also provides a function to test whether a given lock is locked or unlocked: **ustestlock()**. **ustestlock()** returns 1 if the lock is set, and 0 if the lock is not set.

Syntax

```
#include <ulocks.h>
int ustestlock (ulock_t lock);
```

A process can call **usunsetlock()** on a lock that is either not locked or locked by another process. In either case, the lock is unlocked. Double tripping—

that is, calling a set lock function twice with the same lock—is also permissible. The caller blocks until another process unsets the lock.

Using Barriers

Barriers provide a convenient way of synchronizing parallel processes on multiprocessor systems. To use a barrier, you must first allocate one by calling **new_barrier()**, which returns a pointer to an initialized barrier structure. You must then communicate this pointer to the other processes (for example, by placing it in a data structure in shared memory). To arrange a rendezvous, have each process call **barrier()**, passing it the pointer to the barrier structure and a numerical argument specifying the number of processes to wait for. Each process blocks until the last process calls **barrier()**. Barriers are always busy-wait, so they aren't suitable for use on single-processor systems.

Syntax

```
#include <ulocks.h>
barrier_t *new_barrier (usptr_t *handle);
void barrier (barrier_t *b, unsigned n);
void free_barrier (barrier_t *b);
void init_barrier (barrier_t *b);
```

The **free_barrier()** function releases the resources associated with a barrier. The **init_barrier()** function restores a barrier to its original state, as if it had just been allocated by **new_barrier()**.

Using IRIX Shared Memory

Allocating shared memory from a shared arena is much like the regular process of allocating memory using the **malloc()** and **free()** library routines.

Syntax

```
#include <ulocks.h>
#include <malloc.h>
void *usmalloc (size_t size, usptr_t *handle);
void usfree (void *ptr, usptr_t *handle);
void *usrealloc (void *ptr, size_t size, usptr_t *handle);
```

```
void *uscalloc (size_t nelem, size_t elsize, usptr_t *handle);
int usmallopt (int cmd, int value, usptr_t *handle);
struct mallinfo usmallinfo (usptr_t *handle);
```

Memory allocated using **usmalloc()** and related functions can be accessed by all processes that have joined the shared arena specified by *handle*. Other than that, these functions operate in the same way as their single-threaded cousins.

Exchanging the First Datum

Once you've established a shared arena, it's frequently useful to communicate the location of some object in that arena to another process using the arena. For example, one process might create a data structure in shared memory, and pass the address of the data structure to other processes using the arena. The shared arena has a special one-word area for storing such data. This area is accessed using the calls **usputinfo(), usgetinfo(),** and **uscasinfo().** The first two are fairly self-explanatory: **usputinfo()** puts a word of data into the storage area, and **usgetinfo()** returns the current value in the storage area. **uscasinfo()** is explained later.

Syntax

```
#include <ulocks.h>
void usputinfo (usptr_t *handle, void *info);
void *usgetinfo (usptr_t *handle);
```

The following program fragment initializes an arena, allocates space for a data structure, and places the address of the data structure in the storage area. The exact contents of the data structure are left as an exercise for the reader:

```
#include <stdio.h>
#include <ulocks.h>
  /* ... */
  usptr_t *arena;
  char *arenafile = "/usr/tmp/testarena";
  struct controlStruct *control;
  arena = usinit(arenafile);
  if (arena == NULL) {
    /* error */
  }
```

```
  control = usmalloc(sizeof(struct controlStruct), arena);
  if (control == NULL) {
    /* error */
  }
  usputinfo(arena, control);
```

The next program fragment accesses the same shared arena created by the previous program fragment, and obtains the address of the previously allocated data structure:

```
#include <stdio.h>
#include <ulocks.h>
  /* ... */
  usptr_t *arena;
  char *arenafile = "/usr/tmp/testarena";
  struct controlStruct *control;
  arena = usinit(arenafile);
  if (arena == NULL) {
    /* error */
  }
  control = usgetinfo(arena);
  if (control == NULL) {
    /* error */
  }
```

The simple technique outlined above works fine if you know which process is going to create the arena (for example, if program #1 creates the arena before *fork*ing and *exec*ing program #2). However, if several processes are started independently and they all try to access the same arena, the situation becomes more complicated. You may want one of the processes to set up data structures for the others to use. In this instance, IRIX provides an atomic compare-and-swap operator, **uscasinfo()**, which compares the current value in the storage area with a specified value. If these two are equal, **uscasinfo()** changes the value in the storage area to a second specified value.

Syntax

```
#include <ulocks.h>
int uscasinfo (usptr_t *arena, void *oldinfo, void *newinfo);
```

If the value in the storage area is equal to *oldinfo*, **uscasinfo()** changes the value to *newinfo* and returns 1. Otherwise, **uscasinfo()** leaves the value untouched and returns 0. Usually, **uscasinfo()** is called with *oldinfo* equal to 0. The value in the storage area is 0 if no one has placed any data in it yet.

The following code example illustrates a race condition free sequence for a process to attach to an arena. The code expects to be supplied a routine to initialize the control structure that will behave the same way in all the processes that use this code segment and a `register_process` routine that will set up specific structures for the specific process. It is expected that at process termination, the file will be deleted. This is why specific state needs to be considered for this condition.

```
#include <stdio.h>
#include <ulocks.h>
    /* ... */
usptr_t *arena;

char                * arenafile = <name of a file>;
struct controlStruct  * control;
struct controlStruct  * control2;
int                   attempts;

attempts = 20;
while ( attempts -- ) {

    arena = usinit( arenafile );

    /* Make sure that the arena was opened successfully               */
    if ( ! arena ) {
        /* error */
        return ERROR;
    }
    /* Is there a head in the arena yet?            */
    control = ( controlStruct * ) usgetinfo( arena );

    if ( ! control ) {
        control = ( controlStruct * ) usmalloc( sizeof( * control ), arena );

        if ( ! control ) {
            /* error */
            return ERROR;
        }

        /* initialize only the bare minimum of the control structure    */
        control->ushandle = arena;
        control->unlinked = 0;  /* application should set this flag when */
                                /* the last process detaches from arena  */
        control->startlock = usnewlock( arena );
```

```
    if ( ! uscsetlock( control->startlock, 1 ) ) {
        /* I must be able to set the lock that I allocated but didn't*/
        /* error - Internal, should never get here !                 */
        return ERROR;
    }

    doitagain:      /* easiest way to handle this weird case is with*/
                    /* a goto.  Many object to this style but it works*/

    /* compare and swap the control structure                        */
    if ( ! uscasinfo( arena, 0, control ) ) {

        control2 = ( controlStruct * ) usgetinfo( arena );

        if ( ! control2 ) {
            /* uscasinfo failed due to some exception (page fault   */
            /* interrupt etc).  We need to try it again.            */
            goto doitagain;
        }

        /* Free data allocated by this process.                      */
        usfreelock( control->startlock, arena );
        usfree( control, arena );
        control = control2;

    } else {

        /* This process will now initialize the rest of the control */
        /* structure.                                                */

        initialize_control( control );
        handle = register_process( control );

        usunsetlock( control->startlock );

        return handle;
    }
}

ussetlock( control->startlock );

/* In case the initializing process fails or the file gets unlinked */
/* before I get registered check the unlinked flag.                 */
if ( ! control->unlinked ) {
    handle = register_process( control );
```

```
        usunsetlock( control->startlock );

        return handle;
    }

    /* The file got unlinked before I could register.                  */
    /* lets try again - first detach myself.                           */

    usunsetlock( control->startlock );
    usdetach( arena );

} /* while ( 1 ) */

/* file keeps getting unlinked by another process.  Very unlucky.      */
/* Process failed to attach to arena.                                  */

return ERROR;
```

Attaching to an arena asynchronously also requires detaching asynchronously. The following code illustrates how a process can detach itself gracefully from the arena attached in the example above. It assumes that the application provides a routine that will de-register a process from the arena structures. The application needs to determine the condition of no more registered processes.

```
ussetlock( control->startlock );
deregister_pocess( control );
if ( no_registered_processes( control ) ) {
    remove( arenafile );
    control->unlinked = 1;
}

usunsetlock( control->startlock );
usdetach( arena );
```

You can see, IRIX shared memory is very flexible and provides a low overhead solution to IPC problems.

Chapter 2

File and Record Locking

This chapter explains how to lock and unlock files and parts of files from within a program.

Chapter 2

File and Record Locking

This chapter describes how you can use file and record locking capabilities. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

This chapter describes file and record locking, and includes these topics:

- "An Overview of File and Record Locking" presents an introduction to locking mechanisms.
- "Terminology" defines some common terms.
- "File Protection" covers using access permissions, locking files, and getting lock information.
- "Selecting Advisory or Mandatory Locking" describes mandatory locking and record locking across systems.

An Overview of File and Record Locking

Mandatory and advisory file and record locking are available on many current releases of the UNIX system. The intent of these capabilities is to provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi-user applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, on the other hand, the standard I/O subroutines and I/O system calls enforce the locking protocol.

In this way, at the cost of a little efficiency, mandatory locking double-checks the programs to avoid accessing the data out of sequence.

The reference pages for the **fcntl(2)** system call, the **lockf(3)** library function, and *fcntl*(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Terminology

Before discussing record locking mechanisms, first consider a few terms.

Record
: A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. Such structure may be imposed by the programs that use the files.

Cooperating Processes
: Processes that work together in some well-defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term “process” is used interchangeably with “cooperating process” to refer to a task obeying such protocols.

Read (Share) Locks
: These locks are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This may be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Lock
: These are used to gain complete control over sections of files. When a write lock is in place on a record, no other

process may read- or write-lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking
: A form of record locking that does not interact with the I/O subsystem (which includes **creat(**2**)**, **open(**2**)**, **read(**2**)**, and **write(**2**)**). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file is controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking
: A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat(**2**)**, **open(**2**)**, **read(**2**)**, and **write(**2**)** system calls. If a record is locked, then access to that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

The access permissions for each UNIX file control who may read, write, or execute the file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the access permissions for a file. Note that if the permissions for a directory allow anyone to write in the directory, then files within that directory may be removed even by a user who does not have read, write, or execute permission for those files. Any information that is

worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, only protects the portions of the files that are locked. Other parts of the files can be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a database. This can be done easily by setting the set-group-ID bit (see *chmod*(1)) of the database accessing programs. The files can then be accessed by a known set of programs that obey the record-locking protocol. An example of such file protection, although record locking is not used, is the *mail*(1) command. In that command only the owning user and the *mail* command can read and write the unread mail files.

This section covers the following topics:

- "Opening a File for Record Locking"
- "Setting a File Lock"
- "Setting and Removing Record Locks"
- "Getting Lock Information"
- "Deadlock Handling"

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be used, then the file must be opened with at least read access; likewise for write locks and write access.

In the example that follows, we open a file for both read and write access:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
int fd;    /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
```

```
    extern void exit(), perror();

    /* get database file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
}
```

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

Several ways exist to set a lock on a file. These methods depend upon how the lock interacts with the rest of the program. Questions of portability and performance exist. Two methods are given: using the **fcntl(**2**)** system call and using the */usr/group* standards-compatible **lockf(**3**)** library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero and going until the end of the maximum file size. This point is beyond any real end-of-file so that no other lock can be placed on the file. To set such a lock, set the size of the lock to zero. Here is a sample code fragment using the **fcntl()** system call:

```
#include <fcntl.h>
#define MAX_TRY 10
int try;
struct flock lck;

try = 0;
/* set up the record locking structure, the address of which
 * is passed to the fcntl() system call. */
```

```
lck.l_type = F_WRLCK; /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L; /* until the end of the file address space */
/* Attempt locking MAX_TRY times before giving up. */
while (fcntl(fd, F_SETLK, &lck) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other error cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            (void) sleep(2);
            continue;
        }
        (void) fprintf(stderr,"File busy try again
later!\n");
        return;
    }
    perror("fcntl");
    exit(2);
}
...
```

This piece of code tries to lock a file. The lock is attempted several times until one of the following events occurs:

- the file is locked
- an error occurs
- the program exceeds MAX_TRY and gives up

To perform the same task using the **lockf()** function, use code like this:

```
#include <unistd.h>
#define MAX_TRY 10
int try;
try = 0;

/* make sure the file ptr is at the beginning of file. */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up. */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other error cases in which
```

```
         * you might try again.
         */
        if (++try < MAX_TRY) {
            sleep(2);
            continue;
        }
        (void) fprintf(stderr,"File busy try again
later!\n");
        return;
    }
    perror("lockf");
    exit(2);

}
...
```

It should be noted that the **lockf()** example appears to be simpler, but the **fcntl()** example exhibits additional flexibility. Using the **fcntl()** method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf()** merely sets write (exclusive) locks; an additional system call (**lseek()**) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We now try to solve an example problem of dealing with two records (which may be either in the same file or in different files) that must be updated simultaneously so that other processes get a consistent view of the information they contain. This type of problem occurs, for example, when updating the inter-record pointers in a doubly linked list.

To deal with multiple locks, consider the following questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy for the case in which you cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- end the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If processes exist with pending write locks that are waiting for the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is reset with the new lock type. Because the */usr/group* **lockf()** function does not have read locks, lock promotion is not applicable to that call.

The code below shows an example of record locking with lock promotion:

```
struct record {
    .
    .            /* data portion of record */
    .
    long prev;    /* index to previous record in the list */
    long next;    /* index to next record in the list */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there are
 * read locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 * Set a write lock on "this".
 * Return index to "this" record.
 * If any write lock is not obtained:
 * Restore read locks on "here" and "next".
```

```
 * Remove all other locks.
 * Return a -1.
 */
long set3lock (this, here, next)
long this, here, next;
{
    struct flock lck;

    lck.l_type = F_WRLCK;    /* setting a write lock */
    lck.l_whence = 0;        /* offset l_start from beginning of file */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* promote lock on "here" to write lock */
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "this" failed;
         * demote lock on "here" to read lock.
         */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "next" failed; demote lock on "here" to read lock,... */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* ...and remove lock on "this".  */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1) /* cannot set lock, try again or quit */
    }

    return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command was used instead, the **fcntl()** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the **lockf()** function. Since there are no read locks, all (write) locks are referenced generically as locks.

```
/* Lock promotion using lockf(3).
 * When this routine is entered it is assumed that there are
 * no locks on "here" and "next".
 * If locks are obtained:
 * Set a lock on "this".
 * Return index to "this" record.
 * If any lock is not obtained:
 * Remove all other locks.
 * Return a -1.
 */

#include <unistd.h>

long set3lock (this, here, next)
long this, here, next;
{

    /* lock "here" */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* lock "this" */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "this" failed.  Clear lock on "here". */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
    }

    /* lock "next" */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "next" failed.  Clear lock on "here", */
        (void) lseek(fd, here, 0);
```

```
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        /* and remove lock on "this". */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1); /* cannot set lock, try again or quit */
    }

    return (this);
}
```

Locks are removed in the same manner as they are set; only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and only affects locks that were placed by the unlocking process. The unlock only affects the section of the file defined in the previous example by *lck*. It is possible to unlock or change the type of lock on a subsection of a previously set lock; this may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. To find this information, set up a lock as in the previous examples and use the F_GETLK command in the **fcntl()** call. If the lock passed to **fcntl()** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl()**. That is, the lock data passed to **fcntl()** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, *l_pidf* and *l_sysid*, that are only used by F_GETLK. (For systems that do not support a distributed architecture the value in *l_sysid* should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl()** using the F_GETLK command is not blocked by another process' lock, then the *l_type* field is changed to F_UNLCK and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if several read locks occur over the same segment, only one of these is found.

```
struct flock lck;

/* Find and print "write lock" blocked segments of file. */
(void) printf("sysid pid type start length\n");
lck.l_whence = 0;
lck.l_start = 0L;
lck.l_len = 0L;
do {
    lck.l_type = F_WRLCK;
    (void) fcntl(fd, F_GETLK, &lck);
    if (lck.l_type != F_UNLCK) {
        (void) printf("%5d %5d %c %8d %8d\n",
                      lck.l_sysid,
                      lck.l_pid,
                      (lck.l_type == F_WRLCK) ? 'W' : 'R',
                      lck.l_start,
                      lck.l_len);
        /* if this lock goes to the end of the address
         * space, no need to look further, so break out.
         */
        if (lck.l_len == 0)
            break;
        /* otherwise, look for new lock after the one
         * just found.
         */
        lck.l_start += lck.l_len;
    }
} while (lck.l_type != F_UNLCK);
```

fcntl() with the F_GETLK command always returns correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf()** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. Here is a routine using **lockf()** to test for a lock on a file:

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
```

```
    switch (errno) {
    case EACCES:
    case EAGAIN:
        (void) printf("file is locked by another process\n");
        break;
    case EBADF:
        /* bad argument passed to lockf */
        perror("lockf");
        break;

    default:
        (void) printf("lockf: unknown error <%d>\n", errno);
        break;
    }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by *l_start*, when using an *l_whence* value of 1. If both the parent and child process set locks on the same file, possibly a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf()** function call as well and is a result of the */usr/group* requirements for record locking.

If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This results in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl()** system call with an *l_whence* value of 0 or 2. This makes the locking function atomic, so processes sharing file pointers can lock without difficulty.

Deadlock Handling

A certain level of deadlock detection / avoidance is built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf()** call. This deadlock detection is only valid for processes that are locking files or records on a single system.

Deadlocks can potentially occur only when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call fails and sets *errno* to the deadlock error number.

If a process wishes to avoid using the system's deadlock detection, it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

This section covers the following topics:

- "Mandatory Locking"
- "Record Locking Across Multiple Systems"

Mandatory locking is not recommended for reasons stated in the next section, "Mandatory Locking." Whether or not locks are enforced by I/O system calls is determined at the time the calls are made, by the state of the permissions on the file (see *chmod*(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory.

Mandatory enforcement can be assured by code like this:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
...
if (stat(filename, &buf) < 0) {
    perror("program");
    exit (2);
}
/* get currently set mode */
mode = buf.st_mode;
/* remove group execute permission from mode */
mode &= ~(S_IEXEC>>3);
/* set 'set group id bit' in mode */
```

```
mode |= S_ISGID;
if (chmod(filename, mode) < 0) {
    perror("program");
    exit(2);
}
...
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

Use the *chmod*(1) command to set a file to have mandatory locking. This can be done with the command:

```
IRIS% chmod +l filename
```

The *ls* command shows this setting when you ask for the long listing format:

```
IRIS% ls -l filename
-rw---l--- 1 abc other 1048576 Dec 3 11:44 filename
```

Mandatory Locking

Some points to remember about mandatory locking:

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all necessary pieces before any I/O begins. Thus, advisory locking is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking Across Multiple Systems

In a UNIX environment, the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of them, or on yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection and avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process hold record locks on only a single system at any given time for the deadlock mechanism to be effective.

If a process needs to maintain locks over several systems, it is suggested that the process avoid the sleep-when-blocked features of **fcntl()** or **lockf()** and that the process maintain its own deadlock detection. If the process uses the sleep-when-blocked feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Conclusion

Record locking has been added to UNIX System V to ease the development of small to mid-range database systems. The implementation is compatible with the published standards of */usr/group* to this date. Minor differences in operation may exist; they were made for correctness, or to include things not covered by the published standards.

Chapter 3

Working With Fonts

This chapter describes how install, add, and use fonts on Silicon Graphics Computer systems.

Chapter 3

Working With Fonts

This chapter describes how to work with fonts on Silicon Graphics computers. It begins with an introduction to fonts and digital typography. Then it explains which fonts are available and how to install additional fonts. It also covers how to download outline fonts in the Type 1 format to a PostScript printer.

This chapter contains these sections:

- "Font Basics" defines fonts and provides some general background information.
- "Using Fonts with the X Window System" discusses some of the X Window System's most useful font utilities.
- "Installing and Adding Font and Font Metric Files" explains how to install and add font files and font metric files for system-wide use.
- "Downloading a Type 1 Font to a PostScript Printer" explains how to download a Type 1 font to a PostScript® printer.

Font Basics

Fonts are collections of characters. A font contains the information about the shape, size, and position of each character in a character set. That information is needed by those programs which process characters, such as editing, word-processing, desktop-publishing, multimedia, titling, and pre-press application programs. Almost all software components in a computer system use fonts to display messages, prompts, titles, and so forth.

Computers from Silicon Graphics are called digital computers, because discrete voltage levels are used to represent the binary digits 0 and 1. Binary digits are used to represent all other information stored in a digital computer, including fonts. Digital typography deals with the style,

arrangement, and appearance of typeset matter in digital systems. If you want to use font and font metric files to correctly typeset text on a digital computer, you need to know some basics about digital typography. This section contains a brief introduction to fonts and digital typography. You may want to read a book on typography for more in-depth information.

This section covers the following topics:

- "Terminology" introduces a few basic terms.
- "How Resolution Affects Font Size" describes horizontal and vertical resolution, pixels, and bitmap fonts.
- "Font Names" explains the 14-part font name.
- "Writing Programs That Need to Use Fonts" covers X programs, DPS programs, and IRIS GL and IRIS GL/X programs.

Terminology

Before discussing how to use fonts, consider these few terms.

Typography

Typography is the art and technique of working with type. In traditional typography, the term type refers to a piece of wood or metal with a raised image of a character or characters on its upper face. Such pieces of wood or metal are assembled into lines and pages which are printed by a letterpress process. What typographers do with type is called typesetting or composition. Type can also refer to the images produced by using such pieces of wood or metal.

Traditional typesetting is seldom used today. In modern typography, the term type usually refers to the images produced on typesetting or composition systems, which do not use wooden or metal type, such as photo and digital composition systems. The typography on a digital system, such as a digital computer, is called digital typography.

Digital typography is based on a hierarchy of objects called characters, fonts, and font families or typefaces. Numeric values or measurements related to those objects can be divided into character metrics, font metrics, and font family or typeface metrics. Sometimes all information

about a font family or typeface is stored in a set of font files, but sometimes metric information for a set of font files is stored in a separate file called the font metric file.

Characters
: A character is a graphical or mathematical representation of a glyph. Letters, digits, punctuation marks, mathematical symbols, and cursors are examples of glyphs.In a *bitmap* font, the shape of a character is usually represented by a rectangular bitmap. In an *outline* font, the shape of a character is usually represented by a mathematical description of its outline.

Fonts
: A font is a set of characters. A distinction exists between a base and composite font. A *base font* is a set of characters of the same size and style. A *composite font* is composed of base fonts with various attributes. Characters in a base font usually match each other in size, style, weight, and slant because their shape, size, position, and spacing have been carefully designed by a skilled font designer.

Font Family or Typeface
: A professional font designer usually creates an entire *font family* or *typeface*, rather than a single font. A base font family or typeface is a set of base fonts with the same style or design. A composite font family or typeface is composed of base font families or typefaces. A base font family or typeface can consist of bitmap fonts in certain sizes, a scalable font that can be used to produce bitmap fonts in different sizes, or both.

How Resolution Affects Font Size

The images on some output devices, such as laser printers and some types of video monitors, are created by drawing small dots or pixels (picture elements). The number of dots or pixels that can be drawn per unit of length in a horizontal direction is called the *horizontal resolution*, while the number of dots or pixels that can be drawn per unit of length in a vertical direction is called the *vertical resolution*. The most commonly used unit of measure for resolution is the number of dots per inch (dpi). The size of each dot or pixel decreases as the resolution of the output device increases and vice versa; therefore, dots and pixels are device-dependent units of measure.

To display the resolution of your video monitor, enter this command:

```
xdpyinfo | grep resol
```

You should get a response similar to this:

```
resolution: 96x96 dots per inch
```

If you draw all of the characters in a font in the same place (without advancing), you will get a composite image of those characters. If you then draw smallest rectangle that encloses that image, you will get the *bounding box* for that font. The size of a font is usually measured in the vertical direction. That size is usually not smaller than the height of a font bounding box, but it can be greater than that height. It may include additional vertical spacing that is considered part of the font design.

Typographers use small units of measure called *points* to specify font size. A point is approximately equal to 1/72 of an inch. The exact value is 1/72.27 (0.013837) of an inch or 0.351 mm.

A point is a device-independent unit of measure. Its size does not depend on the resolution of an output device. A 12-point font should have approximately the same size on different output devices, regardless of the resolution of those devices.

If the resolution of an output device is equal to 72 dots per inch (dpi), the size of a dot or pixel is approximately equal to the size of a point. If the resolution of an output device is greater than 72 dpi, the size of a dot or pixel is smaller than the size of a point, and vice versa. You can use the following formula to calculate a pixel size from a point size:

```
pixel-size = point-size x device-resolution / 72.27
```

A bitmap font is usually designed for a particular resolution. That font has the point size specified by its designer only when it is used on an output device whose resolution matches the resolution for which that font was designed. This is because a font designer specifies a fixed bitmap for each character. If a pixel is smaller than a point, characters will be smaller than intended, and vice versa.

Font Names

When a font is designed, it is assigned a name such as *Courier Oblique*. This font belongs to a font family called *Courier*, which includes:

- Courier
- Courier Bold
- Courier Bold Oblique
- Courier Oblique

The size of a font is usually not specified for scalable fonts, because they can be scaled to any specified size. Bitmap fonts are usually designed in specific sizes. They are referred to by names such as 12-point Courier or 10-pixel Courier Bold.

When the PostScript page description software language was developed by Adobe Systems, the spaces embedded in font names were replaced with dashes. PostScript font names look like this:

```
Courier
Courier-Bold
Courier-BoldOblique
```

The X Consortium specified 14-part font names for the X Window System.

Figure 3-1 shows an example 14-part name for a bitmap font, with each part labeled. In this Point sizes in X font names are specified in *decipoints* (tenths of a point).

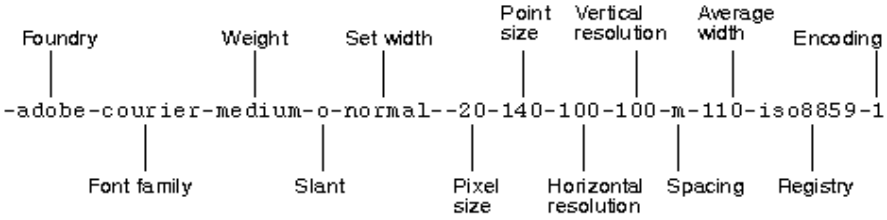

Figure 3-1 X Window System Font Name

For a detailed description of X font names, see the recommended X Window System documentation.

Writing Programs That Need to Use Fonts

You can write different types of programs for Silicon Graphics computers, for example, X, Display PostScript (DPS), IRIS GL, OpenGL, and mixed-model programs. Some of your programs need fonts.

How a program accesses font files depends on the program type:

- X programs access fonts by calling X font functions, such as *XListFonts()* and *XLoadFont()*.
- DPS programs access fonts by calling X and DPS functions, or by using PostScript.
- IRIS GL and IRIS GL / X mixed-model programs usually access fonts by calling font management (fm) functions from the IRIS GL Font Manager library (*fmenumerate()* and *fmfindfont()* for example).

Most fonts are installed when you install the X Window System (X11 Execution Environment). Some fonts are installed with other software components, such as DPS, Impressario™ and IRIS Showcase™. Japanese fonts are installed when you install the Japanese Language System. However, most fonts are shared by the X Window System, DPS (which is an extension of the X Window System), IRIS GL Font Manager, and other software components.

To maintain compatibility and portability, it is best not to access font files directly from an application program because font formats, font names, font contents, and the location of font directories may change. Your program should use the Application Programming Interfaces (APIs) specified for the X Window System, DPS, and IRIS GL Font Manager, or call even higher level functions for the 2D and 3D text available from toolkits such as Inventor™ and Performer.

Using Fonts with the X Window System

This section describes how to use fonts with the X Window System. The X Window System has several font utilities. This section covers a few of the most useful utilities and includes:

- “Getting a List of Font Names and Font Aliases”explains using the *xlsfonts* command.
- “Viewing Fonts” describes the *xfd* command.
- “Getting the Current X Font Path” covers the *xset* command.
- “Changing the X Font Path” explains the *xset fp* command.

For a complete description of the utilities, refer to your X Window System documentation.

Getting a List of Font Names and Font Aliases

To find out which font names and font aliases are known to the X Window System, enter this command:

```
xlsfonts > /tmp/fontlist
```

The resulting file, *fontlist*, contains entries such as:

```
-adobe-courier-bold-o-normal--0-0-0-0-m-0-iso8859-1
-adobe-courier-bold-o-normal--14-100-100-100-m-90-iso8859-1
-sgi-screen-medium-r-normal--14-140-72-72-m-70-iso8859-1
screen14
```

The first entry is an example of a 14-part X name for an outline (scalable) font. Numeric parts of font names are set to zero for outline fonts, because those fonts can be scaled to various sizes. The second and third entries are examples of 14-part X font names for bitmap fonts, while the last entry is an alias for the third entry. An X or DPS program can get a list of available fonts by calling *XListFonts()* or the function *XListFontsWithInfo()*.

Viewing Fonts

To see what a particular font looks like, use the command *xfd*, and specify a font name or font alias from the file *fontlist* by using the option *-fn*. For example, to display the 14-point Adobe Courier Bold font, enter:

```
xfd -fn -adobe-courier-bold-r-normal--14-140-75-75-m-90-iso8859-1
```

To request a Utopia Regular font scaled to the size of 28 points, enter:

```
xfd -fn -adobe-utopia-medium-r-normal--0-280-0-0-p-0-iso8859-1
```

You can use wild cards (*) to indicate that any value is acceptable for parts of an X font name. For example, enter:

```
xfd -fn "-*-courier-bold-r-normal--14-140-75-75-m-90-iso8859-1"
```

to indicate that xfd can use a Courier Bold font from any foundry. Enclose the X font name on a command line in single or double quotes when that name contains wild cards or embedded space characters.

The *xfd* command displays all characters in the specified font, as shown in Figure 3-2.

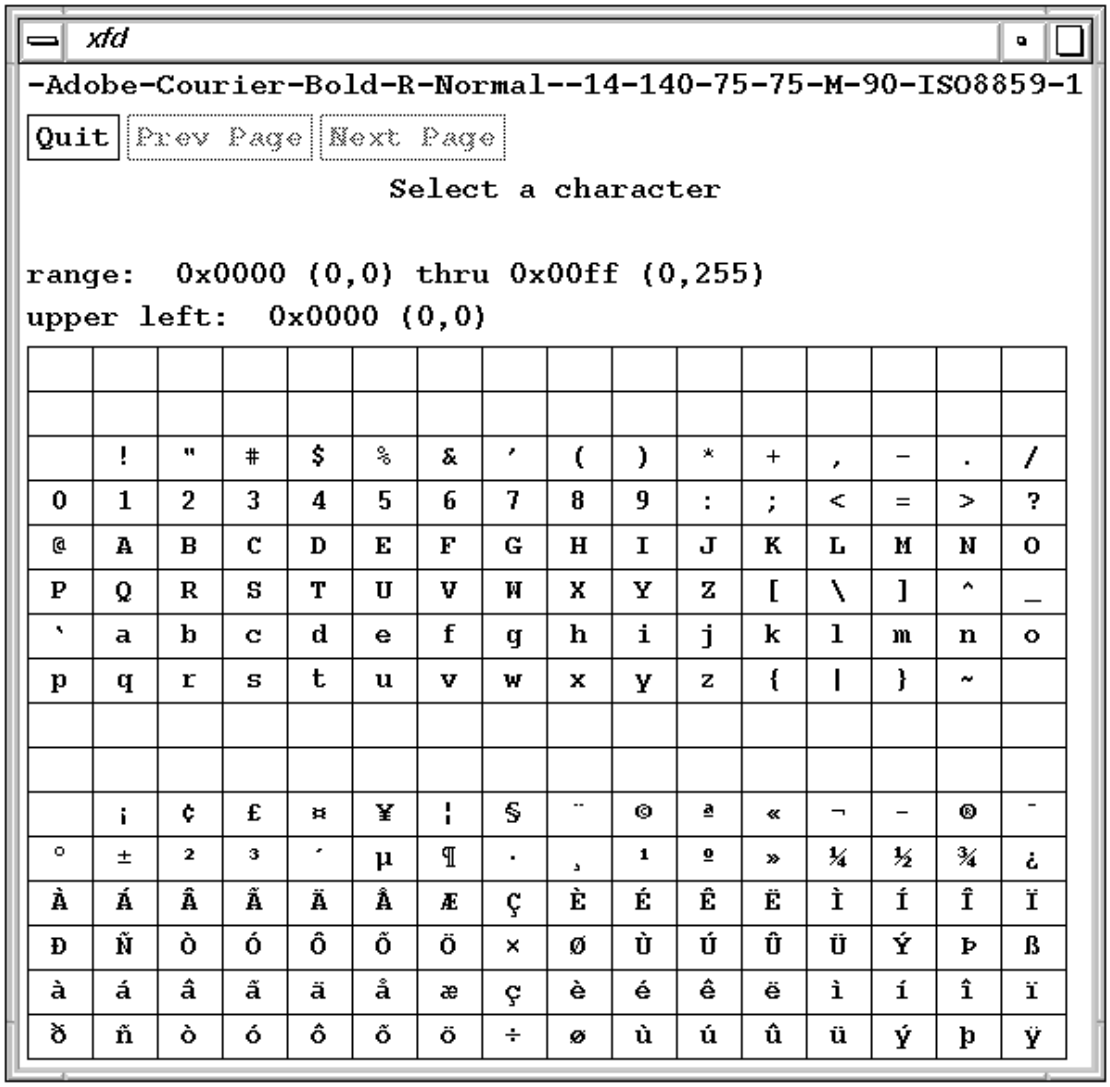

Figure 3-2 Sample Display from *xfd*

To open a shell window that uses a certain font, enter:

`xwsh -fn` *font-name*

Getting the Current X Font Path

To display the current X font path, enter this command:

```
xset q
```

In addition to other information, the *xset* utility displays font path information that may look like this:

```
Font Path:
/usr/lib/X11/fonts/100dpi/,/usr/lib/X11/fonts/75dpi/,
/usr/lib/X11/fonts/misc/,/usr/lib/X11/fonts/Type1/,
/usr/lib/X11/fonts/Speedo
```

The X Window System checks the resolution of your video monitor. If that resolution is closer to 75 dpi than 100 dpi, it puts the directory 75dpi ahead of the directory 100dpi in the X font path.

Changing the X Font Path

You can change the default X font path by using the option *fp* on an *xset* command line. For example, enter:

xset fp=*newpath*

This command changes the X font path to the new font path (*newpath*).

Installing and Adding Font and Font Metric Files

This section explains where the various types of font and font metric files are installed by default, and how you can add one of your font or font metric files to the IRIX operating system.

This section describes the following topics:

- “Installing Font and Font Metric Files” covers getting and changing the current X font path.
- “Adding Font and Font Metric Files” details adding a bitmap and outline font, and adding a font metric file.

Installing Font and Font Metric Files

By default, bitmap font files are installed in these directories: */usr/lib/X11/fonts/100dpi*, */usr/lib/X11/fonts/75dpi*, and */usr/lib/X11/fonts/misc*. The *100dpi* directory contains bitmap fonts designed for the screen resolution of 100 dpi. The *75dpi* directory contains bitmap fonts designed for the screen resolution of 75 dpi. The *misc* directory contains miscellaneous other fonts.

By default, outline font files in the Type 1 format are installed in the directory */usr/lib/DPS/outline/base*.

By default, font files in the directories listed above are used by the X Window System, DPS, IRIS GL Font Manager, and other software components.

The X Window System accesses Type 1 font files with symbolic links in the directory */usr/lib/X11/fonts/Type1*.

By default, outline font files in the Speedo™ format are installed in the directory */usr/lib/X11/fonts/Speedo*. Font files in this directory are currently used only by the X Window System.

By default, Adobe Font Metric (AFM) files are installed in the directory */usr/lib/DPS/AFM*.

There is one font metric file per typeface. Font metric files are primarily used by text-processing and desktop-publishing programs to, for example, generate PostScript code for a specified document.

Adding Font and Font Metric Files

When you purchase fonts or obtain a font that is in the public domain, you may need to add that font to your system or printer in order to use it. Adobe Systems donated bitmap, outline, and font metric files for the Utopia font family to the X Consortium. This section shows how the font and font metric files for Utopia Regular were added to the IRIX operating system. Other font and font metric files can be added in a similar way.

You must log in as root to make any changes to X font directories. Before you make any changes to any IRIX directory, you should make a copy of its

contents so that you can restore that directory if anything goes wrong. For example, your font files may not be in the right format, and they may interfere with the access of Silicon Graphics font files. You should keep a log of the changes you make, and mention those changes when you report a problem with font files to Silicon Graphics; otherwise, it may be very difficult or impossible for other people to reproduce any problems that you might report.

Adding a Bitmap Font

As an example, we show how Utopia Regular bitmap fonts were added to IRIX. Other fonts can be added in a similar way.

To add the Utopia bitmap fonts to the X Window System, Display PostScript, and IRIS GL Font Manager, follow the steps below.

1. Log in as root.
2. Look at the names of existing bitmap font files and try to match these names when you specify new font names. For example, Adobe provided two sets of Utopia Regular bitmap font files that were designed for the resolutions of 100 and 75 dpi. These files were in the extended Bitmap BDF 2.1 format.

   The names of the bitmap files were:

   `UTRG_10.bdf through UTRG_24.bdf`

   When these fonts were added to IRIX, the names were changed to:

   `utopiaR10.bdf through utopiaR24.bdf`

   to match the names of other X bitmap font files.
3. Convert the BDF font files to Portable Compiled Format (PCF) font files.

   BDF font files are text (ASCII) files. You can think of them as source font files. You could put BDF font files into an X font directory, but people usually use binary font formats such as the PCF (*.pcf*) or compressed PCF format (*.pcf.Z*), so you should covert new fonts to one of these formats.

   To convert a BDF font file to a PCF font file, enter a command such as:

   `bdftopcf -o` *file-name*`.pcf` *file-name*`.bdf`

Note: If you have used the *bdftopcf* command before, you may not have specified the name of the output file, but the command now requires that you enter the name of the output file.

You can compress a PCF file by entering a command such as:

`compress` *file-name*`.pcf`

That should give you a file called:

file-name`.pcf.Z`

Most bitmap font files shipped by Silicon Graphics are in the compressed PCF format to save on disk space.

4. Put the bitmap font files for 100 dpi in the directory: */usr/lib/X11/fonts/100dpi*.

   You can tell the resolution for which a font was designed by the name of the directory in which the font designer stored the font files, or by the information in the header of a bitmap font file. In a BDF 2.1 font file, the horizontal (*x*) and vertical (*y*) resolution are specified in the X font name. They are also specified after the point size as the second and third numeric values in a SIZE entry. For example, the entry:

   ```
   SIZE 8 100 100
   ```

   indicates an 8-point font that was designed for the horizontal and vertical resolution of 100 dpi.

   For example, when 100-dpi Utopia Regular bitmap font files were added to IRIX, they were moved to the directory */usr/lib/X11/fonts/100dpi*.

5. Similarly, put the bitmap font files for 75 dpi in the directory */usr/lib/X11/fonts/75dpi*, and put other bitmap fonts into the directory */usr/lib/X11/fonts/misc*.

   For example, when 75-dpi Utopia Regular bitmap font files were added to IRIX, they were placed in the directory */usr/lib/X11/fonts/75dpi*.

6. For most font families shipped by Silicon Graphics, there is one entry per font family in the file:

   /usr/lib/X11/fonts/ps2xlfd_map

The same entry is used for both bitmap and outline fonts. A typical entry in this file looks like this:

Utopia-Regular -adobe-utopia-medium-r-normal--0-0-0-0-p-0-iso8859-1

For each Japanese font family shipped by Silicon Graphics, there is an entry in the file:

/usr/lib/X11/fonts/ps2xlfd_map.japanese

The entries are used by the IRIS Font Manager to map PostScript and other short font names to corresponding X font names. The IRIS Font Manager does not use any bitmap fonts that do not have an entry in those files.

If you add your own (local) bitmap or outline fonts, you should put an entry for each font family in a file called:

/usr/lib/X11/fonts/ps2xlfd_map.local

If that file does not exist, log in as root, and create it. That way your entries will not disappear when you upgrade the system software on your machine.

7. Enter the command:

   mkfontdir

   to create a new *fonts.dir* (fonts directory) file in that directory. For example, when Utopia Regular fonts were added to the directories 100dpi and 75dpi, the command *mkfontdir* was executed in both of those directories.

8. Enter:

   xset fp rehash

   to tell the X Window System that it should check again which fonts are available.

9. To check whether the fonts you added are known to the X Window System, enter:

   xlsfonts > /tmp/fontlist

   The names of the fonts you added should appear on the list of font names and aliases produced by *xlsfonts*.

Bitmap fonts should now be added to the X Window System and the IRIS GL Font Manager. Since DPS needs both outline and bitmap fonts for each supported typeface, it first checks which outline fonts are stored in the directory */usr/lib/DPS/outline/base*. Then it looks for the corresponding bitmap fonts in other X font directories. It ignores all other bitmap fonts. Therefore, DPS ignores the bitmap fonts you added, until you add the corresponding outline fonts.

Make sure that there is no overlap between your entries and the entries in other *ps2xlfd_map** files.

If you do not want to use long X font names, you can specify short aliases for those names. Silicon Graphics software engineers use a file called *fonts.alias* to specify aliases for their fonts. There may be a *fonts.alias* file in an X font directory. For example, see the file *fonts.alias* in the directory:

/usr/lib/X11/fonts/100dpi

A typical font alias looks like this:

```
fixed -misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso8859-1
```

If you want to specify your own (local) font aliases in a font directory, you should specify them in a file called *fonts.alias.local* in that directory. If that file does not exist, log in as root, and create it. That way your entries will not disappear when you upgrade the system software on your machine.

Adding a Font Metric File

AFM files are primarily used by application programs—for example, to generate PostScript code for a specified document. Follow these steps to add a font metric file, follow the steps below.

1. Log in as root.
2. Put Adobe Font Metric files in the directory */usr/lib/DPS/AFM*.

   For example, Adobe provided the Utopia Regular font metric file *UTRG____.AFM*.

The name of an AFM file should also match the PostScript font name. When this font was added to IRIX, the name of the file *UTRG____.AFM* was changed to *Utopia-Regular*, and it was put in the directory */usr/lib/DPS/AFM*.

Adding an Outline Font

To add the Utopia Regular outline font to the X Window System, Display PostScript, and the IRIS GL Font Manager, follow the steps below.

1. Log in as root.
2. Look at the names of existing outline font files in the directory */usr/lib/DPS/outline/base*. Display PostScript requires that the name of each outline font file match the PostScript font name specified in the `/FontName` entry in the header of that outline font file. For example, if you enter:

   `grep /FontName Courier-Bold`

   in the directory */usr/lib/DPS/outline/base*, you get:

   ```
   /FontName /Courier-Bold def
   ```

   You should put only Adobe text (ASCII) Type 1 font files or compatibles into that directory, not binary Type 1 font files or Type 3 font files. Display PostScript can handle Type 3 font files, but the X Window System and IRIS GL Font Manager cannot. You should put into that directory only those outline font files that look like the font files that are already in that directory.

   If you have a binary Type 1 font file (a *.pfb* file), you must convert it into a text (ASCII) Type 1 font file before you can use it on a Silicon Graphics system. To convert *.pfb* files to *.pfa* files, you can use the program *pfb2pfa* from the subsystem *print.sw.desktop* (shipped with IRIX operating system version 5.3 and higher).

   For example, Adobe provided the Utopia Regular outline font file *UTRG____.pfa*, which is an outline font file in the Type 1 format, a special PostScript program that specifies a scalable outline font.

   To find the PostScript font name for this font, enter:

   `grep /FontName UTRG____.pfa`

You should get the response:

```
/FontName Utopia-Regular def
```

This response tells you that the PostScript font name for this font is Utopia-Regular.

When this font was added to IRIX, the name of the file *UTRG____.pfa* was changed to *Utopia-Regular*.

3. Put the file *Utopia-Regular* in the directory */usr/lib/DPS/outline/base*, because that outline font is in the Type 1 format. If you have an outline font in the Speedo format, put it in the directory:

   ```
   /usr/lib/X11/fonts/Speedo
   ```

4. To add the Utopia Regular font and font metric files to Display PostScript, enter:

   ```
   /usr/bin/X11/makepsres -o /usr/lib/DPS/DPSFonts.upr
   /usr/lib/DPS/outline/base /usr/lib/DPS/AFM
   ```

   You should now be able to access the font file you added via Display PostScript.

5. For most font families shipped by Silicon Graphics, there is one entry per font family in the file:

   ```
   /usr/lib/X11/fonts/ps2xlfd_map
   ```

   as described in “Adding a Bitmap Font.” The same entry is used for both bitmap and outline fonts.

   If you add your own (local) bitmap or outline fonts, you should put an entry for each font family in the file called:

   ```
   /usr/lib/X11/fonts/ps2xlfd_map.local
   ```

   You can use entries in the file *ps2xlfd_map* as templates for entries in the file *ps2xlfd_map.local*.

   If the file *ps2xlfd_map.local* does not exist, log in as root, and create it.

   You should now be able to access the font you added via the IRIS GL Font Manager.

6. Display PostScript is an extension of the X Window System. To add an outline font in the Type 1 format to the rest of the X Window System, in any directory, enter the commands:

   ```
   type1xfonts
   xset fp rehash
   ```

   This recreates symbolic links in the directory */usr/lib/X11/fonts/Type1* that point to outline font files in the directory */usr/lib/DPS/outline/base*, and instructs the X Window System to check which fonts are available.

7. To check whether the outline fonts you added are known to the X Window System, enter:

   ```
   xlsfonts > /tmp/fontlist
   ```

   The entries for the outline fonts you added should appear on the list of font names and aliases produced by *xlsfonts*.

Downloading a Type 1 Font to a PostScript Printer

Some outline fonts are usually built into a PostScript printer. You can find out which fonts are known to the PostScript interpreter in your printer by sending the following file to that printer:

```
%!
% Produce a list of available fonts
/f 100 string def
/Times-Roman findfont 12 scalefont setfont
/y 700 def
72 y moveto
FontDirectory {
   pop f cvs show 72 /y y 13 sub def y moveto
} forall
showpage
```

Utopia fonts are not usually built into PS printers. If you try to print a document that requires a Utopia font on a PS printer that does not have that font, a warning message about the replacement of a missing font with a Courier font is sent to the file */usr/spool/lp/log* on the machine to which that PS printer is attached.

You can download a Type 1 font to a PS printer in either of the following two ways:

- You can insert a Type 1 font file at the beginning of the PostScript file that needs that font. You should have a statement that starts with:

  ```
  %!
  ```

  Put this statement at the beginning of your PS file. If you have two such lines, delete the second one.

  When you download a font this way, the font is available only while your print job is being processed.

- You can make a copy of a Type 1 font file, and then insert the statement:

  ```
  serverdict begin 0 exitserver
  ```

  after the first group of comment statements (lines that start with %) if no password has been specified for your printer; otherwise, you should replace `0` in the above statement with the password for your printer. Then you should send the edited file to your printer.

  When you download a font this way, the warning message:

  ```
  %%[ exitserver: permanent state may be changed ]%%
  ```

  is sent to the file */usr/spool/lp/log* on the machine to which the printer is attached.

  The permanent state of the printer is not really changed. Downloaded fonts disappear when you reset the printer by switching its power off and on. If there is not enough memory for additional fonts, you receive a message about a Virtual Memory (VM) error, and the font is not downloaded.

  If you again send the program that produces a list of available fonts to your printer, you should see the PostScript names of the fonts you downloaded on that list.

Chapter 4

Internationalizing Your Application

This chapter explains how to create an application that can be adapted for use in different countries.

Chapter 4

Internationalizing Your Application

Internationalization is the process of generalizing an application so that it can easily be customized—or *localized*—to run in more than one language environment. You can provide internationalized software to produce output in a user's native language, format data (such as currency values and dates) according to local standards, and tailor software to a specific culture.

This chapter describes how to create such an application. It contains the following major sections:

- "Overview" presents an introduction to internationalization and defines some common terms.
- "Additional Reading on Internationalization" describes how to use existing locales or define new ones to choose the environment (language, cultural, character set, etc.) in which an application runs.
- "Character Sets, Codesets, and Encodings" describes various ways of encoding characters, the traditional ASCII being just one of these.
- "Cultural Items" discusses the ways in which different cultures affect the way a string can be viewed, for example in outputting or collating.
- "Strings and Message Catalogs"describes how to create and use catalogs of messages to send diagnostic information to users in various locales.
- "Internationalization Support in X11R6" describes internationalization support provided by X11, Release 6 (X11R6) (including features from X11R5).
- "User Input" discusses the translation of keyboard event into programmatic character strings for a variety of keyboards.
- "GUI Concerns" discusses internationalizing applications that use Graphical User Interfaces (GUIs)
- "Popular Encodings" presents some common non-ASCII encodings.

For a list of ISO 3166 country names and abbreviations, see Appendix A, "ISO 3166 Country Names and Abbreviations." You can find detailed information about fonts in Chapter 3, "Working With Fonts." Also, you can find additional information about internationalizing an application in the *Indigo Magic Desktop Integration Guide*.

Overview

Internationalized software can be made to produce output in a user's native language, to format data (such as dates and currency values) according to the user's local customs, and to otherwise make the software easier to use for users from a culture other than that of the original software developer. As computers become more widely used in non-American cultures, it becomes increasingly more important that developers stop relying on the conventions of American programming and the English language in their programs; this chapter provides information on how to make your applications more widely accessible.

This section presents the following topics:

- "Some Definitions" covers locales, internationalization, localization, nationalized software, and multilingual software.
- "Areas of Concern in Internationalizing Software" points out a few concerns to watch for when internationalizing your software.
- "Standards" covers standard-compliant features.
- "Internationalizing Your Application: The Basic Steps" lists the procedure to used when internationalizing an icon.
- "Additional Reading on Internationalization" provides references you can consult for additional information about internationalization.

Some Definitions

This section defines some of the terms used in this chapter.

Locale

Locale refers to a set of local customs that determine many aspects of software input and output formatting, including natural language, culture, character sets and encodings, and formatting and sorting rules. The locale of a program is the set of such environmental parameters that are currently selected. For information on the method for selecting locales, see "Additional Reading on Internationalization," below.

Internationalization (I18n)

Internationalization is the process of making a program capable of running in multiple locales without recompiling. To put it another way, an internationalized program is one that can be easily localized without changing the program itself. (See "Localization (L10n)," below, for an explanation of the term "localization.")

Note: The word "internationalization" consists of an 'i' followed by 18 letters followed by an 'n.' It is thus commonly abbreviated "i18n." "I18n" is pronounced "internationalization," not "eye-eighteen-enn."

A program written for a specific locale may be difficult to run in a different environment. Rewriting such a program to operate in each desired environment would be tedious and costly.

Your goal as a developer should thus be to write *locale-independent* programs, programs that make no assumptions about languages, local customs, or coded character sets. Such internationalized applications can run in a user's native environment following native conventions with native messages, without recompiling or relinking. A single copy of an internationalized program can be used by a world of different users.

Localization (L10n)

Localization is the act of providing an internationalized application with the environment and data it needs to operate in a particular locale. For example,

adding German system messages to IRIX is a part of localizing IRIX for the German locale.

Note: Localization is often abbreviated “l10n.”

Nationalized Software

Nationalized programs run in only one language and are governed by one set of customs; in other words, in a nationalized program the locale is built into the application. Even if the application doesn’t use ASCII or English, as long as it is a single-language program it is nationalized, not internationalized. Most older UNIX programs can be thought of as being nationalized for the United States.

Consider two applications, *hello* and *bonjour*. The application *hello* always produces the output

```
Hello, world.
```

and *bonjour* always produces

```
Bon jour, tout le monde.
```

Neither *hello* nor *bonjour* are internationalized; they are both nationalized.

There are no special requirements for writing or porting nationalized applications, whether they are text or graphics programs. Terminal-based programs work on suitable terminals, including internationalized terminal emulators. “Suitable” means that the terminal supports any necessary fonts and understands the encoding of the application output. Graphics programs simply do as they have always done. Applications using existing interfaces to operate in non-English or non-ASCII environments should continue to compile and run under an internationalized operating system.

Multilingual Software

A *multilingual* program is one that uses several different locales at the same time. Examples are described in “Multilingual Support” on page 147.

Areas of Concern in Internationalizing Software

Few developers will have to pay attention to more than a few items described in this section. Most will need to catalog their strings. Some will need to use library routines for character sorting or locale-dependent date, time, or number formatting. A few whose applications use the eighth bit of 8-bit characters inappropriately will need to stop doing so. The few applications that do arithmetic to manipulate characters will need to be cleaned up. Some GUI designers will have to spend just a little more time thinking. But for the large majority of developers, there isn't much to do.

The information presented in the following sections addresses internationalization issues pertinent to a developer; some sections, however, may not be applicable to your applications.

Standards

IRIX internationalization includes these standards-compliant features, among others:

- ANSI C and POSIX (ISO 9945-1): Locale
- *X/OPEN Portability Guide, Issue 3* (XPG/3): XPG/3 message catalogs, interpretation of locale strings
- AT&T UNIX™ System V Release 4: Multi-National Language Support (MNLS) message catalogs
- X11R5 and X11R6: Input methods, text rendering, resource files

Internationalizing Your Application: The Basic Steps

To internationalize your icon, follow these steps:

1. Call **setlocale()** as soon as possible to put the process into the desired locale. See "Setting the Current Locale" on page 142 for instructions.
2. Make your application 8-bit clean. (An application is 8-bit clean if it does not use the high bit of any data byte to convey special information.) See "Eight-Bit Cleanliness" on page 149 for instructions.

3. If you're writing a multilingual application, you must do one of two things. Either

   - fork, and then call **setlocale()** differently in each process

   or

   - call **setlocale()** repeatedly as necessary to change from language to language

   See "Multilingual Support" on page 147 for more information.

4. Use WC or MB characters and strings to allow for more than one byte per character (this is needed for Asian languages, which often require two or even four bytes per character). See "Character Representation" on page 151 for more information.

5. Do not rely on ASCII and English sorting rules. Locale-specific collation should be performed with **strcoll()** and **strxfm()**. (These are table-driven functions; the tables are supplied as part of locale support.). See "Collating Strings" on page 156 for more information.

6. Use the **localeconv()** function to find out about numeric formatting data. (Format of simple numbers differs from locale to locale.) See "Specifying Numbers and Money" on page 158 for more information.

7. Use **strftime()** to format dates and times (**strftime()** gives a host of options for displaying locale-specific dates and times.) See "Formatting Dates and Times" on page 159 for more information.

8. Avoid arithmetic on character values. Use the macros in ctype.h macros to determine various kinds of information about a given character. (These macros are table-driven and are therefore locale-sensitive.) If you prefer, you can use the functions that correspond to these macros instead. "Character Classification and ctype" on page 160 provides more detailed information on these macros and functions.

9. If you do your own regular expression parsing and matching, use the XPG/3 extensions to traditional regular expression syntax for internationalized software. See "Regular Expressions" on page 161 for more information.

10. Where possible, use the XPG/3, rather than the MNLS interface in order to maximize portability. See "Strings and Message Catalogs" on page 161 for more information.

11. Provide a catalog for the your locale. See “SVR4 MNLS Message Catalogs” on page 166 for more information.
12. Internationalize FTR strings. See “Internationalizing File Typing Rule Strings” on page 169 for more information.
13. Use message catalogs for **printf()** format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. See “Variably Ordered Referencing of printf() Arguments” on page 171 for more information.
14. If you’re using Xlib, initialize Xlib’s internationalization state after calling **setlocale()**. See “Initialization for Xlib Programming” on page 175 for more information.
15. Specify a default fontset suitable for the default locale. Make sure that the application accepts localized fontset specifications via resources (or message catalogs) or command line options. See “Fontsets” on page 176 for more information.
16. Use X11R5 and X11R6 text rendering routines that understand multibyte and wide character strings, not the X11R4 text rendering routines **XDrawText()**, **XDrawString()**, and **XDrawImageString()**. See “Text Rendering Routines” on page 178 for more information.
17. Use X11R5 and X11R6 MB and WC versions of width and extents interrogation routines. See “New Text Extents Functions” on page 178 for more information.
18. If you are writing a toolkit text object, or if you can’t use a toolkit to manage event processing for you, then you have to deal with input methods. Follow the instructions in “User Input” on page 180.
19. Use resources to label any object that employs some sort of text label. Your application’s app-defaults file should specify every reasonable string resource. See “X Resources for Strings” on page 196 for more information.
20. Use dynamic layout objects that calculate layout depending on the natural (localized) size of the objects involved. Some IRIS IM widgets providing these services are XmForm, XmPanedWindow, and XmRowColumn. See “Dynamic Layout” on page 198 for more information. If you can’t use dynamic layout objects, refer to “Layout” on page 197 for instructions.

21. Make sure that all icons and other pictographic representations used by your application are localizable. See “Icons” on page 199 for more information.

Additional Reading on Internationalization

For more information on internationalization, refer to:

- O’Reilly Volume 1, *Xlib Programming Manual*
- *X Window System*, by Robert Scheifler and Jim Gettys.
- *X/Open Portability Guide*
- *OSF/Motif Style Guide*

Locales

An internationalized system is capable of presenting and receiving data understandably in a number of different formats, cultures, languages and character sets. An application running in an internationalized system must indicate how it wants the system to behave. IRIX uses the concept of a locale to convey that information.

A process can have only one locale at a time. Most internationalization interfaces rely on the locale of the current process being set properly; the locale governs the behavior of certain library routines.

This section covers the following topics:

- “Setting the Current Locale” explains categories, locales, strings, location of locale-specific data, and locale naming conventions.
- “Limitations of the Locale System” describes multilingual support, misuses of locales, and encoding.

Setting the Current Locale

Applications begin in the *C* locale. (*C* is the name used to indicate the system default locale; it usually corresponds to American English.) Applications

should therefore call *setlocale()* as soon as possible to put the process into the desired locale. The syntax for *setlocale()* is:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

The call almost always looks either like this:

```
if (setlocale(LC_ALL, "") == NULL)
    exit_with_error();
```

or like this:

```
if (setlocale(LC_ALL, "") == NULL)
    setlocale(LC_ALL, "C");
```

Details of the two parameters are given in the next two sections.

Category

Applications need not perform every aspect of their work in the same locale. Although this approach is not recommended, an application could (for example) perform most of its activities in the English locale but use French sorting rules. You can use locale categories to do this kind of locale-mixing. (Mixing locale categories is not the same as multilingual support—see "Multilingual Support.")

The *category* argument is a symbolic constant that tells *setlocale()* which items in a locale to change. Table 4-1 lists the available category choices.

Table 4-1 Locale Categories

Category	Affects
LC_ALL	All categories below
LC_COLLATE	Regular expressions, *strcoll()*, and *strxfrm()*
LC_CTYPE	Regular expressions and ctype routines (such as *islower()*)
LC_MESSAGES[a]	*gettxt()*, *pfmt()*, and *nl_langinfo()*

Table 4-1 (continued) Locale Categories

Category	Affects
LC_MONETARY	localeconv()
LC_NUMERIC	Decimal-point character for formatted I/O and nonmonetary formatting information returned by *localeconv()*
LC_TIME	*ascftime()*, *cftime()*, *getdate()*, and *strftime()*

a. LC_MESSAGES is supported by SVR4 but isn't required by XPG/3.

Categories correspond to databases that contain relevant information for each defined locale. The locations of these databases are given in the "Location of Locale-Specific Data" on page 146.

Locale

The *setlocale()* function attempts to set the locale of the specified category to the specified locale. You should almost always pass the empty string as the *locale* parameter to conform to user preferences.

On success, *setlocale()* returns the new value of the category. If *setlocale()* couldn't set the category to the value requested, it returns NULL and does not change locale.

The Empty String

An empty string passed as the *locale* parameter is special. It specifies that the locale should be chosen based on environment variables. This is the way a user specifies a preferred locale, and that preference should almost always be honored. The variables are checked hierarchically, depending on category, as shown in Table 4-2; for instance, if the category is LC_COLLATE, an empty-string locale parameter indicates that the locale should be chosen based on the value of the environment variable LC_COLLATE—or, if that value is undefined, the value of the environment

variable LANG, which should contain the name of the locale that the user wishes to work in.

Table 4-2 Category Environment Variables

Category	First Environment Variable	Second Environment Variable
LC_COLLATE	LC_COLLATE	LANG
LC_CTYPE	LC_CTYPE	LANG
LC_MESSAGES	LC_MESSAGES	LANG
LC_MONETARY	LC_MONETARY	LANG
LC_NUMERIC	LC_NUMERIC	LANG
LC_TIME	LC_TIME	LANG

Specifying the category LC_ALL attempts to set each category individually to the value of the appropriate environment variable.

If no non-null environment variable is available, *setlocale()* returns the name of the current locale.

Nonempty Strings in Calls to *setlocale()*

Here are the possibilities for specifying a nonempty string as the *locale* parameter:

NULL string
: Specifying a locale value of NULL—not the same as the empty string—causes *setlocale()* to return the name of the current locale.

"C"
: Specifying a locale value of the single-character string "C" requests whatever locale the system uses as a default. (Note that this is a string and not just a character.)

Other nonempty strings
: Requesting a particular locale to be used by specifying its name. Overrides any user preferences; this should only be done with good reason.

Location of Locale-Specific Data

Except for XPG/3 message catalogs, locale-specific data (that is, the "compiled" files containing the collation information, monetary information, and so on) are located in */usr/lib/locale/*locale/category, where *locale* and *category* are the names of the locale and category, respectively. For example, the database for the LC_COLLATE category of the French locale *fr* would be in */usr/lib/locale/fr/LC_COLLATE*.

There will probably be multiple locales symbolically linked to each other, usually in cases where a specific locale name points to the more general case. For example, */usr/lib/locale/En_US.ascii* might point to */usr/lib/locale/C*.

Locale Naming Conventions

A locale string is of the form:

language[*_territory*[*.encoding*]][*@modifier*]...

where:

- *language* is the two-letter ISO 639 abbreviation for the language name.
- *territory* is the two-uppercase-letter ISO 3166 abbreviation for the territory name. (For a list of these abbreviations, see the table in Appendix A, "ISO 3166 Country Names and Abbreviations.")
- *encoding* is the name of the character encoding (mapping between numbers and characters). For western languages, this is typically the codeset, such as 8859-1 or ASCII. For Asian languages, where an encoding may encode multiple codesets, the encodings themselves have names, such as UJIS or EUC (these encodings are described later in this section). "Character Sets, Codesets, and Encodings" on page 149 discusses codesets and encodings.
- *modifier*s are not actually part of the locale name definition; they give more specific information about the desired localized behavior of an application. For example, under X11R5 or X11R6, a user can select an input method with modifiers. (To use the *xwnmo* Input Method server provided by Silicon Graphics, for example, add *@im=_XWNMO* to the locale string.) No standards exist for this part of a locale string.

Language data is implementation specific; databases for the language *en* (English) might contain British cultural data in England and American cultural data in the United States. If other than the default settings are required, the territory field may be used. For example, the above cases could be more strictly defined by setting LANG to *en_EN* or *en_US*. Full rigor might lead to *en_EN.88591* for England (the locale encoding specification for ISO 8859-1 is “88591”) and *en_US.ascii* for the USA.

ANSI C has defined a special locale value of *C*. The *C* locale is guaranteed to work on all compliant systems and provides the user with the system’s default locale. This default is typically American English and ASCII, but need not be. POSIX has also defined a special locale value, *POSIX*, which is identical to the *C* locale.

The length of the locale string may not exceed NL_LANGMAX characters (NL_LANGMAX is defined in */usr/include/limits.h*). However, XPG/3 recommends that this string (not counting modifiers) not exceed 14 characters.

Limitations of the Locale System

Multilingual Support

There can only be one locale at a time associated with any given process in an internationalized system. Therefore, although multilingual applications—which give the appearance of using more than one locale at a time—can be created, internationalization does not provide inherent support for them. Here are two examples of multilingual programs:

- An application creates and maintains windows on four different displays, operated by four different users. The program has a single controlling process, which is associated with only one locale at any given time. However, the application can switch back and forth between locales as it switches between users, so the four users may each use a different locale.
- In a sophisticated editing system with a complex user interface, a user may wish to operate the interface in one language while entering or editing text in another. For instance, a user whose first language is German may wish to compose a Japanese document, using Japanese

input and text manipulation, but with the user interface operating in German. (There is no standard interface for such behavior.)

In writing a multilingual application, the first task is identifying the locales for the program to run in and when they apply. (There is no standard method for performing this task.) Once the application has chosen the desired locales, it must either one of the following:

- fork, and then call *setlocale()* differently in each process
- call *setlocale()* repeatedly as necessary to change from language to language

Misuse of Locales

The LANG environment variable and the locale variables provide the freedom to configure a locale, but they do not protect the user from creating a nonsensical combination of settings. For example, you are allowed to set LANG to *fr* (French) and LC_COLLATE to *ja_JP.EUC* (Japanese). In such a case, string routines would assume text encoded in 8859-1—except for the sorting routines, which might assume French text and Japanese sorting rules. This would likely result in arbitrary-seeming behavior.

No Filesystem Information for Encoding Types

The IRIX filesystem does not contain information about what encoding should be associated with any given data. Thus, applications must assume that data presented to an application in some locale is properly encoded for that locale. In other words, a file is interpreted differently depending on locale; there is no way to ask the file what it thinks its encoding is.

For example, you may have created a file while in a Japanese locale using EUC. Later, you might try printing it while in a French locale. The results will likely resemble a random collection of Latin 1 characters.

This problem applies to almost all stored strings. Most strings are uninterpreted sequences of nonzero bytes. This includes, for example, filenames. You can, if you want to, name your files using Chinese characters in a Chinese locale, but the names will look odd to anyone who runs */bin/ls* on the same filesystem using a non-Chinese locale.

Character Sets, Codesets, and Encodings

One major difference between nationalized and internationalized software is the availability in internationalized software of a wide variety of methods for encoding characters. Developers of internationalized software no longer have the convenience of always being able to assume ASCII. Three terms that describe groupings of characters are:

character set An abstract collection of characters.

codeset A character set with exactly one associated numerical encoding for each character. The English alphabet is a character set; ASCII is a codeset.

encoding A set of characters and associated numbers; however, this term is more general than "codeset." A single encoding may include multiple codesets; *Extended Unix Code (EUC*), for instance, is an encoding that provides for four codesets in one data stream.

This section describes these topics:

- "Eight-Bit Cleanliness" explains how to make 8-bit clean characters.
- "Character Representation" discuses multibyte and wide characters.
- "Multibyte Characters" covers using and handling multibyte characters, conversions to constant-size characters, and the number of bytes in a character and string.
- "Wide Characters" explains *wchar* strings, support routines, and conversion to multibyte characters.
- "Reading Input Data" covers nonuser-originated data.

For information on installing and using fonts with an application, refer to Chapter 3, "Working With Fonts."

Eight-Bit Cleanliness

A program is *8-bit clean* if it does not use the high bit of any data byte to convey special information. ASCII characters are specified by the low seven bits of a byte, so some programs use the high bit of a data byte as a flag; such programs are not 8-bit clean. Internationalized programs must be 8-bit clean,

because they cannot expect data to be in the form of ASCII bytes; non-ASCII character sets usually use all eight bits of each byte to specify the character. But a program must go out of its way to manipulate bytes based on the value of the high bit; and since changing data without cause is seldom desirable, most programs are already 8-bit clean.

The old *csh* (before this problem was fixed in the IRIX 5.0 release) was a good example of a program that was not 8-bit clean; it used the high bit in input strings to distinguish aliases from unaliased commands. An effect of this misuse was that *csh* stripped the 8th bit from all characters:

```
echo I know an architect named Mañosa
I know an architect named Maqosa
```

Another example is the specification of Internet messages, which calls for 7-bit data. Thus, if *sendmail* fails to strip the 8th bit from a character prior to sending it, it violates a protocol; if it does strip the bit, it could garble a non-ASCII message (this protocol problem is being addressed).

One of the simplest things to do to remove the American bias from a program is to replace the ASCII assumption with the assumption that the Latin 1 codeset will be used. This approach is not true internationalization, but it can make the application usable in most of Western Europe. Latin 1 uses only one byte per character, unlike some other codesets, so 8-bit clean ASCII software should work without modification using the Latin 1 codeset.

Ensuring that code is 8-bit clean is the single most important aspect of internationalizing software.

Another caveat about 8-bit characters only applies to a particular set of circumstances: if you're not using a multibyte character type (see the next section), you should not declare characters as type signed char. (The default in IRIX C is for char to imply unsigned char.) If you try to cast a signed char to an int (as you must do to use the ctype(3C) functions) and the character's high bit is set (as it may be in an 8-bit character set), the high bit is interpreted as a sign bit and extends into the full width of the int.

Character Representation

Western languages usually require only one byte for each character. Asian languages, however, often require two or even four bytes per character; and some Asian encodings allow a variable number of bytes per character.

The two kinds of encodings that allow more than one byte per character are:

- multibyte (*MB*) characters (which are of variable size)
- wide (*WC* or *wchar*) characters (which are a fixed number of bytes long)

The application developer must decide where to use WC and MB characters and strings:

- Multibyte strings are almost the default: string I/O uses MB, MB code works for ASCII and ISO 8859, and MB characters use less space than do wide characters. However, manipulating individual characters within a multibyte string is difficult.

  Note: Traditional strings are merely a special case of multibyte strings, where every character happens to be one byte long and there is only one codeset. All MB code, including conversion to and from wchars, works for traditional ASCII, or ISO 8859, strings.

- Applications that do heavy string manipulation typically use WC strings for such activity, because manipulating individual WC characters in a string is much simpler than doing the same thing with MB characters. So wide characters are used as necessary to provide programming ease or runtime speed; however, they take up more space than MB characters.

  Note: WC is system dependent—applications should not use it for I/O strings or communication.

Multibyte Characters

A multibyte character is a series of bytes. The character itself contains information on how many bytes long it is. Multibyte characters are referenced as strings (and are therefore of type *char **); before parsing, a string is indistinguishable from a multibyte character. The zero byte is still used as a string (and MB character) terminator.

A string of MB characters can be considered a null-terminated array of bytes, exactly like a traditional string. A multibyte string may contain characters from multiple codesets. Usually, this is done by incorporating special bytes that indicate that the next character (and only the next character) will be in a different codeset. Very little application code should ever need to be aware of that, though; you should use the available library routines to find out information about multibyte strings rather than look at the underlying byte structure, because that structure varies from one encoding to another. For one example of an encoding that allows characters from multiple codesets, see “EUC” on page 202.

Use of Multibyte Strings

Multibyte strings are very easy to pass around. They efficiently use space (both data and disk space), since “extra” bytes are used only for characters that require them. MB strings can be read and written without regard to their contents, as long as the strings remain intact. Displaying MB strings on a terminal is done with the usual routines: *printf()*, *puts()*, and so on. Many programs (such as *cat*) need never concern themselves with the multibyte nature of MB strings, since they operate on bytes rather than on characters; so MB strings are often used for string I/O.

Manipulation of individual characters in an MB string can be difficult, since finding a particular character or position in a string is nontrivial (see “Handling Multibyte Characters,” below). Therefore, it is common to convert to WC strings for that kind of work.

Handling Multibyte Characters

Usually, multibyte characters are handled just like *char* strings. Editing such strings, however, requires some care.

You cannot tell how many bytes are in a particular character until you look at the character. You cannot look at the *n*th character in a string without looking at all the previous *n* - 1 characters, because you cannot tell where a character starts without knowing where the previous character ends. Given a byte, you don’t know its position within a character. Thus, we say the string has *state* or is *context-sensitive*; that is, the interpretation we assign to any given byte depends on where we are in a character.

This analysis of characters is locale-dependent, and therefore must be done by routines that understand locale.

Conversion to Constant-Size Characters

Multibyte characters and strings are convertible to wchars via *mbtowc(*3*)* (individual characters) and *mbstowcs(*3*)* (strings).

How Many Bytes in a Character?

To find out how many bytes make up a given single MB character, use *mblen(*3*)*:

```
#include <stdlib.h>
. . .
size_t n;
int len;
char *pStr;
. . .
len = mblen(pStr, n); /* examine no more than n bytes */
```

It is the application's responsibility to ensure that *pStr* points to the beginning of a character, not to the middle of a character.

The maximum number of bytes in a multibyte character is MB_LEN_MAX, which is defined in *limits.h*. The maximum number of bytes in a character under the current locale is given by the macro MB_CUR_MAX, defined in *stdlib.h*.

How Many Bytes in an MB String?

Since *strlen()* simply counts bytes before the first NULL, it tells you how many bytes are in an MB string.

How Many Characters in an MB String?

When *mbstowcs()* coverts MB strings to WC strings, it returns the number of characters converted. This is the simplest way to count characters in an MB string.

Note: Many code segments that need to deal with individual characters within a string would be better suited by wide character strings. Since counting often involves conversion, such segments are often better served by working with a WC string, then converting back.

Getting the length without performing the conversion is straightforward, but not as simple. *mbtowc()* converts one character and returns the number of bytes used, but returns the same information without conversion if a NULL is passed as the address of the WC destination. Thus:

```
len = mblen(pStr, n);
```

is equivalent to

```
len = mbtowc((wchar_t *) NULL, pStr, n);
```

In fact, *mblen()* calls *mbtowc()* to perform its count. Therefore, counting characters in an MB string without converting would look like this:

```
int cLen;
char *tStr = pStr;

numChars = 0;
cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
while (cLen > 0) {
    tStr += cLen;
    numChars++;
    cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
    if (cLen == -1)
        numChars = cLen; /* invalid MB character */
}
```

Wide Characters

A wide character (*WC* or *wchar*) is a data object of type *wchar_t*, which is guaranteed to be able to hold the system's largest numerical code for a character. *wchar_t* is defined in *stdlib.h*. Under IRIX 4.0.x, *sizeof(wchar_t)* was 1. Under IRIX 5.1, it is 4. All wchars on a system are the same size, independent of locale, encoding, or any other factors.

Uses for *wchar* Strings

The single advantage of WC strings is that all characters are the same size. Thus, a string can be treated as an array, and a program can simply index into the array in order to modify its contents. Most applications' *char* manipulation routines work with little modification other than a type change to *wchar_t*, with appropriate attention to byte count and *sizeof()*.

So, when applications have significant string editing to perform, they typically keep the strings in WC format while doing that editing. Those WC strings may or may not be converted to or from MB strings at other points in the application.

Wide characters are often large and are not as space efficient as multibyte strings. Applications that do not need to perform string editing probably shouldn't use *wchar*s. If an application intends to both maintain and edit large numbers of strings, then the developer needs to make size / complexity trade-off decisions.

Support Routines for Wide Characters

Analogs to the routines defined in *string.h* and *stdio.h* are supplied in *libw.a* and defined in *widec.h*. This includes routines such as *getwchar()*, *putwchar()*, *putws()*, *wscpy()*, *wslen()*, and *wsrchr()*.

Conversion to MB Characters

Wide characters and strings are convertible to MB strings via *wctomb()* and *wcstombs()*, respectively.

Reading Input Data

Input can be divided into two categories: user events and other data. This section deals with nonuser-originated data, which is assumed to come from file descriptors or streams. User events are discussed in "User Input" on page 180.

It is generally fair to assume that unless otherwise specified, data read by an application is encoded suitably for the current locale. Text strings typically are in MB format.

Streams can be read in WC format by using routines defined in *widec.h*.

Cultural Items

This section discusses several aspects of a locale that may differ between locales. It includes these topics:

- "Collating Strings" describes string collation.
- "Specifying Numbers and Money" explains some monetary formats, and the *printf()* and *localeconv()* functions.
- "Formatting Dates and Times" covers using *strftime()* to format of dates and times.
- "Character Classification and ctype" discusses associations between character codes, and using *ctype.h* macros and functions.
- "Regular Expressions" presents information for developers who do their own regular expression parsing and matching.

Collating Strings

Different locales can have different rules governing collation of strings, even within identical encodings.

The Issue

In English, sorting rules are extremely simple: each character sorts to exactly one unique place. Under ASCII, the characters are even in numeric order. However, neither of those statements is necessarily true for other languages and other codesets. Furthermore:

- Sorting order for a language may be completely unrelated to the (numerical) order of the characters in a given encoding.

- Even with a correctly sorted list of the characters in a character set, you may not be able to sort words properly.
- Locales using identically encoded character sets may use very different sorting rules.

Programs using ASCII can do simple arithmetic on characters and directly calculate sorting relationships; such programs frequently rely on truisms such as the fact that:

```
'a' < 'b'
```

in ASCII. But internationalized programs cannot rely on ASCII and English sorting rules. Consider some non-English collation rule types:

- *One-to-Two* mappings collate certain characters as if they were two. For example, the German *ß* collates as if it were "ss."
- *Many-to-One* mappings collate a string of characters as if they were one. For example, Spanish sorts "ch" as one character, following "c" and preceding "d." In Spanish, the following list is in correct alphabetical order: *calle*, *creo*, *chocolate*, *decir.*
- *Don't-Care Character* rules collate certain characters as if they were not present. For example, if "-" were a don't-care character, "co-op" and "coop" would sort identically.
- *First-Vowel* rules sort words based first on the first vowel of the word, then by consonants (which may precede or follow the vowel in question).
- *Primary/Secondary* sorts consider some characters as equals until there is a tie. For example, in French, a, á, à, and â all sort to the same primary location. If two strings (such as "tache" and "tâche") collate to the same primary order, then the secondary sort distinguishes them.
- Special case sorts exist for some Asian languages. For example, Japanese *kanji* has no strict sorting rules. *Kanji* strings can be sorted by the strokes that make up the characters, by the *kana* (phonetic) spellings of the characters, or by other agreed-upon rules.

It should be clear that a programmer cannot hope to collate strings by simple arithmetic or by traditional methods.

The Solution

Locale-specific collation should be performed with *strcoll()* and *strxfm()*. These are table-driven functions; the tables are supplied as part of locale support. The value of LC_COLLATE determines which ordering table to use. *strcoll()* has the same interface as *strcmp()*; it can be directly substituted into code that uses *strcmp()*.

Specifying Numbers and Money

Format of simple numbers differs from locale to locale. Characters used for decimal radix and group separators vary. Grouping rules may also vary. Even though we assume that decimal numbers are universal, there are some eighteen varying aspects of numeric formatting defined by a locale. Many of these are details of monetary formatting.

For example, Germany uses a comma to denote a decimal radix and a period to denote a group separator. English reverses these. India groups digits by two except for the last three digits before the decimal radix. Many locales have particular formats used for money, some of which are shown in Table 4-3:

Table 4-3 Some Monetary Formats

Country	Positive Format	Negative Format
India	Rs1,02,34,567.89	Rs(1,02,34,567.89)
Italy	L.10.234.567	-L.10.234.567
Japan	¥10,234,567	-¥10,234,567
Netherlands	F10.234.567,89	F-10.234.567,89
Norway	Kr10.234.567,89	Kr10.234.567,89-
Switzerland	SFr10,234,567.89	SFr10,234,567.89C

printf()

printf() examines LC_NUMERIC and chooses the appropriate decimal radix. If none is available, it tries to use ASCII period. No further locale-specific

formatting is done directly by *printf()*. However, you may wish to refer to "Variably Ordered Referencing of printf() Arguments," for a way to do your own locale-specific output formatting.

localeconv()

The *localeconv()* function can be called to find out about numeric formatting data, including the decimal radix (inappropriately called *decimal_point*), the grouping separator (inappropriately called *thousands_sep*), the grouping rules, and a great deal of monetary formatting information. Actual use of formatting information other than the decimal radix is left to the application; there aren't any special print routines that produce formatted numbers according to all of *localeconv()*'s data.

Formatting Dates and Times

All of these dates can mean the same thing to different people:

```
92.1.4
4/1/92
1/4/92
```

All of these can mean the same time to different people:

```
2:30 PM
14:30
14h30
```

Dates and times can be easily formatted by using *strftime()*, which gives a host of options for displaying locale-specific dates and times. The *ascftime()* and *cftime()* functions give further options, but should be avoided because they do not conform to ANSI and XPG/3 specifications. The old *asctime()* and *ctime()* functions are now obsolete; use *strftime()* instead. For more information, see the *strftime(*3C*)* reference page.

Character Classification and *ctype*

The *ctype.h* header file is described in the *ctype*(3C) reference page and defines macros to determine various kinds of information about a given character: *isalpha(), isupper(), islower(), isdigit(), isxdigit(), isalnum(), isspace(), ispunct(), isprint(), isgraph(), iscntrl(),* and *isascii().*

The Issue

When programmers knew that a character set was ASCII, some convenient assumptions could be made about characters and letters. It was common for programmers to do arithmetic with the ASCII code values in order to perform some simple operations. For example, raising a character to upper case could be done by subtracting the difference between the code for *a* and the code for *A*. Numeric characters could be identified by inspection: if they fell between *0* and *9*, they were numeric; otherwise, they weren't. You could tell if a character was (for instance) printable, a letter, or a symbol by comparing to known encoding values. Macros for such activity have long been available in *ctype.h*, but lots of programs did character arithmetic anyway. Since character encoding and linguistic semantics are completely independent, such arithmetic in an internationalized program leads to unpleasant results.

Furthermore, characters exist outside of ASCII that break some non-arithmetic assumptions. Consider the German character *ß* which is a lowercase alphabetic character (letter), yet has no uppercase. Consider also French (as written in France), where the uppercase of *é* is *E*, not *É*.

Clearly, the programmer of an internationalized application has no way of directly computing all the character associations that were available in English under ASCII.

The Solution

Strict avoidance of arithmetic on character values should remove any trouble in this area. The macros in *ctype.h* are table-driven and are therefore locale-sensitive. If you think of characters as abstract characters rather than as the numbers used to represent them, you can avoid pitfalls in this area.

Using Functions Instead of Macros

A corresponding function exists for each of the macros in *ctype.h*. To get a function instead of a macro, simply undefine the name. For example:

```
#undef toascii
char (*xlate)(char);
char a, b;
...
xlate = toascii;
a = (*xlate)(b);
```

Regular Expressions

XPG / 3 specifies some extensions to traditional regular expression syntax for internationalized software. Few application developers do their own regular expression parsing and matching, however, so we do not include full details here. Briefly, the extensions provide the ability to specify matches based on:

- character class (such as *alpha*, *digit*, *punct*, or *space*)
- equivalence class (for instance: *a*, *á*, *à*, *â*, *A*, *Á*, *À*, and *Â* may be equivalent)
- collating symbols (allowing you to match the Spanish “ch” as one element because it is a single collating token)
- generalization of range specifications of the form $[c_1\text{-}c_2]$ to include the above

Programmers who process expressions themselves need the full description of internationalized regular expression grammar in Volume 3, Chapter 6, of XPG / 3.

Strings and Message Catalogs

Message catalogs are compiled databases of strings. While a major role of message catalogs is to provide communications text in locale-specific natural language, the strings can be used for any purpose. The idea is that an application uses only strings from a catalog, thus allowing localizers to supply catalogs suitable for a given locale.

Two different and incompatible interfaces to message catalogs exist in IRIX: *MNLS* and *XPG/3*. Developers working on SVR4 or other AT&T code, or related base-system utilities, probably use MNLS. Developers working on independent projects probably use XPG/3. Neither is a solid standard, but XPG/3 is closer to being a standard than MNLS. Thus applications developers who have to choose between the two interfaces are encouraged to use XPG/3 to maximize their portability. XPG/3 seems to be popular in Europe. Asia is leaning toward MNLS.

This section describes string and message catalogs, specifically:

- "XPG/3 Message Catalogs"
- "SVR4 MNLS Message Catalogs"
- "Variably Ordered Referencing of printf() Arguments"

XPG/3 Message Catalogs

The XPG/3 message catalog interface requires that a catalog be opened before it is read, and requires that catalog references specify a catalog descriptor.

Since catalog references include a default to be used in case of failure, applications will work normally without a catalog when in the default locale. This means catalog generation is exclusively the task of localizers. But in order to inform the localizer as to what strings to translate and how they should comprise a catalog, the application developer should provide a catalog for the developer's locale.

Opening and Closing XPG/3 Catalogs

catopen() locates and opens a message catalog file:

```
#include <nl_types.h>
nl_catd catopen(char *name, int unused);
```

The argument *name* is used to locate the catalog. Usually, this is a simple, relative pathname that is combined with environment variables to indicate the path to the catalog (see "XPG/3 Catalog Location" for details). However, the catalog assumes names that begin with " / " are absolute pathnames. Use

of a hard-coded pathname like this is strongly discouraged; it doesn't allow the user to specify the catalog's locale through environment variables.

When an application is finished using a message catalog, it should close the catalog and free the descriptor using *catclose()*:

```
int catclose(nl_catd);
```

Using an XPG/3 Catalog

Catalogs contain sets of numbered messages. The application developer must know the contents of the catalog in order to specify the set and number of a message to be obtained.

catgets() is used to retrieve strings from a message catalog:

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_num, int msg_num,
               char *defaultStr);
```

catgets() retrieves message *msg_num* from set *set_num* from the catalog described by *catd*. If for any reason *catgets()* cannot do this, it returns *defaultStr*. Example 4-1 shows a program which reads the first message from the first message set in the appropriate catalog, and displays the result.

Example 4-1 Reading an XPG/3 Catalog

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

#define SET1      1
#define WRLD_MSG  1

int main(){
    nl_catd msgd;
    char *message;
    setlocale(LC_ALL, "");

msgd = catopen("hw",0);
    message = catgets(msgd, SET1, WRLD_MSG,"Hello, world\n");
    printf(message);
    catclose(msgd);
}
```

The previous example uses *printf()* instead of *puts()* in order to make a point: the format string of *printf()* came from a catalog. Note that:

```
printf(catgets(msgd, set, num, defaultStr));
```

is very different from:

```
printf("%s", catgets(msgd, set, num, defaultStr));
```

because strings in catalogs can contain formatting strings of the kind used by *printf()*. For further discussion of issues relating to this important distinction, see "Variably Ordered Referencing of printf() Arguments."

XPG/3 Catalog Location

XPG/3 message catalogs are located using the environment variable NLSPATH. The default NLSPATH is */nlslib/%L/%N*, where *%L* is filled in by the LANG environment variable and *%N* is filled in by the *name* argument to *catopen()*. NLSPATH can specify multiple pathnames in ordered precedence, much like the PATH variable. A sample NLSPATH assignment:

```
NLSPATH=/usr/lib/locale/%L/%N:/usr/local/lib/locale/%L/%N:/usr/defaults/%N
```

Full details are in the *catopen(*3*)* reference page.

Creating XPG/3 Message Catalogs

Message catalogs are of this general form:

```
$set n comment
a message-a\n
b message-b\n
c message-c\n
$quote "
d " message-d "
$this is a comment
```

Each message is identified by a *message number* and a *set*. Sets are often used to separate messages into more easily usable groups, such as error messages, help messages, directives, and so on. Alternatively, you could use a different set for each source file, containing all of that source file's messages.

"*$set* n" specifies the beginning of set *n*, where *n* is a set identifier in the range from 1 to NL_SETMAX. All messages following the "*$set* n" statement belong to set *n* until either a *$delset* or another *$set* is reached. You can skip set numbers (for example, you can have a set 3 without having a set 2), but the set numbers that you use must be listed in ascending numerical order (and every set must have a number). Any string following the set identifier on the same line is considered a comment.

"*$delset* n" deletes the set *n* from a message catalog.

"*$quote* c" specifies a quote character, *c*, which can be used to surround message text so that trailing spaces or null (empty) messages are visible in a message source line. By default, there is no quote character and messages are separated by newlines. To continue a message onto a second line, add a backslash to the end of the first line:

```
$set 1
1 Hello, world.
2 here is a long \
string.\n
3 Hello again.
n message-text-n
```

Message #2 in set #1 is "here is a long string.\n".

Compiling XPG/3 Message Catalogs

After creating the message catalog sources, you need to compile them into binary form using *gencat*, which has the following syntax:

```
gencat catfile msgfile [msgfile ...]
```

where *catfile* is the target message catalog and *msgfile* is the message source file. If an old *catfile* exists, *gencat* attempts to merge new entries with the old. *gencat* "resolves" set and message number conflicts with new information replacing the old.

The *catfile* then needs to be placed in a location where *catopen()* can find it; see the "XPG/3 Catalog Location" on page 164.

SVR4 MNLS Message Catalogs

There are many ways to use strings from MNLS message catalogs. You can get strings directly and then use them, or you can use output routines that search catalogs.

Specifying MNLS Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified every time a string is needed.

To specify the default message catalog to be used by subsequent calls to MNLS routines that reference catalogs (such as *gettxt()*, *tfmt()*, or *pfmt()*), use *setcat()*:

```
#include <pfmt.h>
char *setcat(const char *catalog);
```

where *catalog* is limited to 14 characters, with no character equal to zero or the ASCII codes for / (slash) or : (colon). *setcat()* doesn't check to see if the catalog name is valid; it just stores the string for future reference. For an example of use, see "Getting Strings from MNLS Message Catalogs," below. The catalog indicated by the string can be found in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Getting Strings from MNLS Message Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified in each reference call. Strings are read from a catalog via *gettxt()*:

```
#include <unistd.h>
char *gettxt(const char *msgid, const char *defaultStr);
```

msgid is a string containing two fields separated by a colon:

msgfilename:*msgnumber*

The *msgfilename* is a catalog name as described previously in the “Specifying MNLS Catalogs” on page 166. For example, to get message 10 from the *MQ* catalog, you could use either:

```
char *str = gettxt("MQ:10", "Hello, world.\n");
```

or

```
setcat("MQ");
str = gettxt(":10", "Hello, world.\n");
```

pfmt()

pfmt() is one of the most important routines dealing with MNLS catalogs, because it is used to produce most system diagnostic messages. *pfmt()* formats like *printf()* and produces standard error message formats. It can usually be used in place of *perror()*. For example,

```
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce, by default (such as when the Mozambique locale is unavailable),

```
ERROR: Permission denied.
```

The syntax of *pfmt()* is:

```
#include <pfmt.h>
int pfmt(FILE *stream, long flags, char *format, ... );
```

The *flags* are used to indicate severity, type, or control details to *pfmt()*. The format string includes information specifying which message from which catalog to look for. Flag details are discussed next in the section “Labels, Severity, and Flags”, and the format is discussed in the “Format Strings for pfmt()” on page 168.

Labels, Severity, and Flags

pfmt() flags are composed of several groups; specify no more than one from each group. Specify multiple flags by using OR.

The groups are:

output format control:
MM_NOSTD, MM_STD

catalog access control:
MM_NOGET, MM_GET

severity: MM_HALT, MM_ERROR, MM_WARNING, MM_INFO

action message specification:
MM_ACTION

pfmt() prints messages in the form *label:severity:text*. *Severity* is specified in the *flags*. The *text* comes from a message catalog (or a default) as specified in the *format*, and the *label* is specified earlier by the application.

In the example above, we get only:

```
ERROR: Permission denied.
```

if no label has been set. Typically, an application sets the label once early in its life; subsequent error messages have the label prepended. For example:

```
setlabel("UX:myprog");
...
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce (by default):

```
UX:myprog: ERROR: Permission denied.
```

For details, consult the *pfmt(*3*)* and *setlabel(*3*)* reference pages.

Format Strings for *pfmt()*

pfmt() format strings are of the form:

[[*catalog*:]*messagenum*:]*defaultstring*

The *catalog* field is in the format described above in the “Specifying MNLS Catalogs” on page 166. *messagenum* is the message number in the catalog to use as the format. *defaultstring* specifies the string to use if the catalog lookup fails for any reason.

An important feature of *pfmt()* is its ability to refer to format arguments in format-specified order just as *printf()* does. See “Variably Ordered Referencing of printf() Arguments” for details.

fmtmsg()

fmtmsg() is a comprehensive formatter using the MNLS catalogs and “standard” formats. You probably won’t need to use it; most applications should get by with *pfmt()*, *gettxt()*, and *printf()*. Consult the *fmtmsg(*3*)* reference page for details.

Putting Strings into a Catalog

An MNLS catalog source file contains merely a list of strings, separated by new lines. For an empty string, an empty line is used. Strings are referenced by line number in the original source file.

Applications access the catalog by line number, so it’s very important not to change the line numbers of existing catalog entries. This means that, when you want to add a new string to an existing catalog source, you should always append it to the end of the file—if you put it in the middle of the file, then you change the line number for subsequent strings.

Tools exist that help you compile MNLS message catalogs. *exstr*, for instance, extracts strings directly from your source code and replaces them with calls to the message retrieval function.

When a file of strings is ready to be compiled, simply run *mkmsgs* and put the results in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Internationalizing File Typing Rule Strings

You can internationalize the strings defined in the LEGEND and MENUCMD rules in the File Typing Rule (FTR) file. To internationalize these rules, precede the string with the following:

:*catalogname*:*msgnumber*:

where *catalogname* is optional and should be a valid MNLS catalog, *msgnumber* is the line number in *catalogname*. If you omit *catalogname*, the *uxsgidesktop* catalog is used by default.

You can use these rules to create your own FTR catalog. For example, an entry looks like this:

```
LEGEND :mycatalog:7:Archive 8mm Tape Drive
```

This entry uses line 7 from the catalog, *mycatalog*, as the LEGEND for this FTR. If *mycatalog* is not available, or line 7 is not accessible from *mycatalog*, "Archive 8mm Tape Drive" is used as the LEGEND.

```
LEGEND :7:Archive 8mm Tape Drive
```

This entry uses line 7 from the *uxsgidesktop* catalog, if available. Otherwise, "Archive 8mm Tape Drive" is used.

The next example,

```
MENUCMD \'mycatalog:9:Eject Tape\' /usr/sbin/eject /dev/tape
```

displays line 9 from *mycatalog*, if available. Otherwise "Eject Tape" is displayed on the menu that pops up when you click on an icon using this FTR.

You can internationalize strings in the command part of MENUCMD and CMD rules by using *gettxt* or any other convenient policy detailed in this section. For example:

```
CMD OPEN xconfirm -t "Tape tool not available"
```

can be internationalized to:

```
CMD OPEN xconfirm -t "'gettxt mycatalog:376 'Tape tool not
available''"
```

In this example, *gettxt* is invoked to access line 376 from the catalog, *mycatalog*, and the string returned by *gettxt* is passed to *xconfirm* for display. If line 376 from *mycatalog* is not accessible, then *gettxt* returns the string "Tape tool not available."

For more information about FTRs, see the *Indigo Magic Desktop Integration Guide*."

Variably Ordered Referencing of *printf()* Arguments

printf() and its variants can now refer to arguments in any specified order. Consider the following scenario: an application has chosen “house” from a list of objects and “white” from a list of colors. The application wishes to display this choice. The code might look like:

```
char *obj, *color;
... /* make choices */ ...
printf("%s %s\n", color, obj);
```

And the *printf()* yields:

```
white house
```

Even once we make sure that *obj* and *color* are localized strings, we are not quite finished. If our locale is Spanish, the *printf()* yields:

```
blanca casa
```

which is wrong; in Spanish, it should be:

```
casa blanca
```

The solution to this problem is *variably ordered referencing* of *printf()* arguments. The syntax of *printf()* format strings has been expanded.

If a format string used to contain %*T* (where *T* represents one of the *printf()* conversion characters), the string can now contain %*D*$*T*. The *T* is the same, but the *D* specifies which argument from the argument list should be used.

This means you can write:

```
printf("2nd parameter is %2$s; the 1st is %1$s", p1, p2)
```

and the *second* parameter is printed *first*, with the first parameter printed second. For example:

```
char *store = "Macy’s";
char *obj = "a cup";

printf("At %1$s, I bought %2$s.\n", store, obj);
printf("I bought %2$s at %1$s.\n", store, obj);
```

This code yields:

```
At Macy's, I bought a cup.
I bought a cup at Macy's.
```

even though the parameters to *printf()* are in the same order.

In English, we are able to come up with strings suitable for either word order; in some other language, we might not be so lucky. Nor can we predict which order such languages might prefer. So the developer has no way of knowing how to create traditional *printf()* format strings suitable for all languages.

Developers should therefore use message catalogs for their *printf()* format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. This means that the localizer has much greater ability to create intelligible text. An internationalized version of the above code appears in the following example.

Example 4-2 Internationalized Code

```
/* internationalized (XPG/3) version */
char *form = catgets(msgd, set, formNum,
                  "At %1$s, I bought %2$s.\n");
char *store = catgets(msgd, set, storeNum, "Macy's");
char *obj = catgets(msgd, set, objNum, "a cup");

printf(form, store, obj);
```

The unlocalized (default) version would produce:

```
At Macy's, I bought a cup.
```

and a localized version might produce:

```
Compré una tasa en Macy's.
```

In practice, variably ordered format strings are only found in message catalogs and not in default strings. The default string usually simply uses the parameters in the order they're given, without the new variable-order format strings.

Internationalization Support in X11R6

X11R6 internationalization support is provided on the X client side; that is, the application must take care of such support instead of relying on the X server. No server changes are necessary, and the protocol is unchanged. Full backward compatibility is preserved, so a new internationalized application can run on an old server.

Note: X11R6 internationalization refers to features in X11R5 and X11R6.

X uses existing internationalization standards to do its internationalization support; there are no X-specific interfaces to set and change locale. Internationalized X applications receive no help from X when attempting multilingual support. No locales or special process states are peculiar to X.

This section explains:

- "Limitations of X11R6 in Supporting Internationalization" discusses vertical text, character sets, and Xlib interface changes.
- "Resource Names" covers encoding of resource names.
- "Getting X Internationalization Started" describes initialization of Xlib and toolkit programming.
- "Fontsets" explains specifying, creating, and using fontsets,
- "Text Rendering Routines" discusses the *XmbDrawText()*, *XmbDrawString()*, and *XmbDrawImageString()* functions.
- "New Text Extents Functions" describes a few new extents-related functions including *XFontSetExtents*.

Limitations of X11R6 in Supporting Internationalization

Since X is locale-independent, there are some limitations on its ability to support internationalization. The X protocol and Xlib specification, together with ANSI C and POSIX restrictions, have led to the following choices being made in X11R6:

Vertical Text

There is no built-in support for vertical text. Applications may draw strings vertically only by laying out the text manually.

Character Sets

In previous releases of X, there was no general support for character sets other than Latin 1. X11R6, however, does allow other character sets.

X11R6 includes the definition of the *X Portable Character Set*, which is required to exist in all locales supported by Xlib. There is no encoding defined for this set; it is only a character set. The set, similar to printable ASCII, consists of these characters:

```
abcdefghijklmnoqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~
<space> <tab> <newline>
```

The *Host Portable Character Encoding* is the encoding of the X Portable Character Set on the Xlib host. This encoding is part of X, and is thus independent of locale—the coding remains the same for all locales supported by the host.

Strings used or returned by Xlib routines are either in the Host Portable Character Encoding or a locale-specific encoding. The Xlib reference pages specify which encodings are used where. Some string constructs (such as *TextProperty*) contain information regarding their own encoding.

Xlib Interface Change

Full use of X11R6's internationalization features means calling some new routines supplied in the X11R6 Xlib. While all old Xlib applications work with the new Xlib, developers should change their code in places. These are described below.

Resource Names

Resource names are compiled into programs. Because of that, their encoding must be known independent of locale. Trying to add a level of indirection here results in a problem: you're always left with something compiled that can't be localized. Resource names therefore use the X Portable Character Set. The names may be anything; at least they'll mean something to the application author. (If the names were numbers, for example, they would be meaningless to everybody.)

Getting X Internationalization Started

Xlib's internationalization state, like that of *libc*, needs to be initialized.

Initialization for Xlib Programming

Initialize Xlib's internationalization state after calling *setlocale()*. Xlib is being initialized, not a server or server-specific object, so a server connection is not necessary:

```
if ( setlocale(LC_ALL, "") == NULL )
    exit_with_error();
if ( ! XSupportsLocale() )
    exit_with_other_error();
if ( XSetLocaleModifiers("") == NULL)
    give_warning();
```

XSetLocaleModifiers() is only required for input. Just as passing an empty string to *setlocale()* honors the user's environment, so does passing an empty string to *XSetLocaleModifiers()*.

Initialization for Toolkit Programming

If you're using Xt (with a widget set such as IRIS IM, Motif, or XaW) then you don't use *setlocale()*. Instead you should use:

```
XtSetLanguageProc(NULL, NULL, NULL)
```

If you're using a toolkit other than Xt, call *setlocale()* as early as possible after execution begins.

Fontsets

In X11R5 and X11R6, unlike previous releases of X, a string may contain characters from more than one codeset. There are several methods for determining which codeset a given character is in; which method is appropriate depends on the locale and the encoding used.

For information on installing and using fontsets with an application, refer to Chapter 3, “Working With Fonts.”

Such multiple-codeset strings usually cannot be rendered using a single font. A *fontset* is a collection of fonts suitable for rendering all codesets represented in a locale’s encoding. A fontset includes information to indicate which locale it was created in. Applications create fontsets for their own use; when a program creates a fontset, it is told which of the requested fonts are unavailable.

Example: EUC in Japanese

To render strings encoded in EUC in Japanese, an application would need fonts encoded in 8859-1, JIS X 208, and JIS X 201. The application doesn’t need to know which characters in a string go with which font, since it doesn’t get into locale specifics. So it creates a fontset that is made from a list of user-specified fonts (under the assumption that the localizer has provided an appropriate list). Rendering is then done using that fontset. The locale-aware rendering system chooses the appropriate fonts for each character being rendered, from the supplied list. You can find additional information about EUC in “Asian Languages.”

Specifying a Fontset

A fontset specification is just a string, enumerating XLFD names of fonts. (See *X Logical Font Description Conventions*, an MIT X Consortium standard.) This string can include wild card characters. For example, a specification of 15-point “fixed” fonts might be:

```
char *fontSetSpecString = "*fixed-medium-r-normal*150*";
```

A particular server might expand this to:

```
-jis-fixed-medium-r-normal--16-150-75-75-c-160-jisx0208.1983-0
```

```
-sony-fixed-medium-r-normal--16-150-75-75-c-80-iso8859-1
-sony-fixed-medium-r-normal--16-150-75-75-c-80-jisx0201.1976-0
```

Specifying the fontset by simply enumerating the fonts is perfectly acceptable:

```
char *fontSetSpecString =
"-jis-fixed-medium-r-normal*150-75-75*jisx0208.1983-0,\
-sony-fixed-medium-r-normal*150-75-75*iso8859-1,\
-sony-fixed-medium-r-normal*150-75-75*jisx0201.1976-0";
```

A German locale would use this fontset with interest only in the ISO font; a Japanese locale might use all three; a Chinese locale would have trouble with this fontset.

The developer should specify a default fontset suitable for the default locale. Furthermore, developers should ensure that the application accepts localized fontset specifications via resources (or message catalogs) or command line options. Localizers are responsible for providing default fontset specifications suitable for their locales.

Creating a Fontset

Creating fontsets in X is simply a matter of providing a string that names the fonts, as described above.

```
XFontSet fontset;
char *base_name;  /* should get from resource */
char **missingCharsetList;
int missingCharsetCount;
char *defaultStringForMissingCharsets;

base_name = "*fixed-medium-r*150*"; /* use resources! */

fontset = XCreateFontSet(display, base_name,
                         &missingCharsetList,
                         &missingCharsetCount,
                         &defaultStringForMissingCharsets);
```

The locale in effect at create time is bound to the fontset. Fontsets are freed with *XFreeFontSet()*.

Using a Fontset

Fontsets are used when rendering text with X11R6 *Xmb* or *Xwc* text rendering routines. These routines are described in "Text Rendering Routines."

Text Rendering Routines

X11R6includes text rendering routines that understand multibyte and wide character strings. These routines are analogs to the X11R4 text rendering routines *XDrawText()*, *XDrawString()*, and *XDrawImageString()*. The old routines continue to operate, but do not take fontsets, and don't know how to handle characters longer than one byte.

- *XmbDrawText()* and *XwcDrawText()* take lists of *TextItems*, each of which contains (among other things) a string. The strings are rendered using fontsets. These routines allow complex spacing and fontset shifts between strings.
- *XmbDrawString()* and *XwcDrawString()* render a string using a fontset. These routines render in foreground only and use the raster operation from the current graphics context.
- *XmbDrawImageString()* and *XwcDrawImageString* also render a string using a fontset. These routines fill the background rectangle of the entire string with the background, then render the string in the foreground color, ignoring the currently active raster operation.

Consult the appropriate reference pages for more details on these routines.

New Text Extents Functions

X11R6 provides MB and WC versions of *width* and *extents* interrogation routines, supplying the maximum amount of space required to draw any character in a given fontset. These routines depend on fontsets to interpret strings and use locale-specific data.

The *XFontSetExtents* structure contains the two kinds of extents a string can have:

```
typedef struct {
    XRectangle max_ink_extent;
    XRectangle max_logical_extent;
} XFontSetExtents;
```

max_ink_extent gives the maximum boundaries needed to render the drawable characters of a fontset. It considers only the parts of glyphs that would be drawn, and gives distances relative to a constant origin. *max_logical_extent* gives the maximum extent of the *occupied space* of drawable characters of a fontset. The occupied space of a character is a rectangle specifying minimum distance from other graphical features; other graphics generated by a client should not intersect this rectangle. *max_logical_extent* is used to compute interline spacing and the minimum amount of space needed for a given number of characters.

Here are descriptions of a few of the new extents-related functions (consult the appropriate reference pages for details):

- *XExtentsOfFontSet()* returns an *XFontSetExtents* structure for a fontset.
- *XmbTextEscapement()* and *XwcTextEscapement()* take a string and return the distance in pixels (in the current drawing direction) to the origin of the next character after the string, if the string were drawn. Escapement is always positive, regardless of direction.
- *XmbTextExtents()* and *XwcTextExtents()* take a string and return information detailing the overall rectangle bounding the string's image and the space the string occupies (for spacing purposes).
- *XmbTextPerCharExtents()* and *XwcTextPerCharExtents()* take a string and return ink and logical extents for each character in the string. Use this for redrawing portions of strings or for word justification. If the fontset might include context-dependent drawing, the client cannot assume that it can redraw individual characters and get the same rendering.
- *XContestDependentDrawing()* returns a Boolean telling whether a fontset might include context-dependent drawing.

User Input

This section has to do with the translation of physical user events into programmatic character strings or special keyboard data (such as “backspace”). This kind of work should be done by toolkits. If you can use a toolkit to manage event processing for you, do so, and blissfully ignore this section. If you are writing a toolkit text object, or are writing a truly extraordinary application, then this section is for you.

This section on user input covers these topics:

- “About User Input and Input Methods” presents an overview of user input and input methods
- “About X Keyboard Support” covers X keyboard support, including keys, keycodes, keysyms, and composed characters.
- “Input Methods (IMs)” describes opening and closing input methods, and IM styles.
- “Input Contexts (ICs)” explains an IM styles, IC values, pre-edit and status attributes, and creating and using ICs.
- “Events Under IM Control” describes differences in processing events under IM control including *XFilterEvent()* and *LookupString* routines.

About User Input and Input Methods

Just as internationalized programs cannot assume that data is in ASCII, they cannot assume that user input will use any specific keyboard. Keyboards change from country to country and language to language; internationalized software should never assume that a certain position on the keyboard is bound to a certain character, or that a given character will be available as a single keystroke on all keyboards.

No useful physical keyboard—not even one specifically designed for multilingual work—could possibly contain a key for every character we would ever wish to type. Certainly there are characters commonly used in other areas of the world that are not present on most USA keyboards. So methods have been invented that provide for input of almost any known character on even the most naïve keyboards. These schemes are referred to as *input methods* (IMs).

Note: IMs should not be confused with the IRIS IM product, the Silicon Graphics port of the OSF/Motif product.

Input methods vary significantly in design, use, and behavior, but there is a single API that developers use to access them. The object is for the application simply to ask for an IM and let the system check the locale and choose the appropriate IM.

Some IMs are complex; others are very simple. The API is designed to be a low-level interface, like Xlib. Usually, only toolkit text object authors must deal with the IM interfaces. However, some applications developers are unable to use toolkit objects, so the concepts are described here.

Reuse Sample Code

A sample program demonstrating some of the concepts in this section is given in Chapter 11 of the *Xlib Programming Manual, Volume One*. Looking carefully at that code may be easier than starting from scratch.

GL Input

The old GL function *qdevice()* has a hard-coded view of a keyboard (see */usr/include/gl/device.h* for details). Some flexibility, particularly for Europe, is available if you queue KEYBD instead of individual keys, but the GL has no general solution to non-ASCII input. There is no supported way to input Chinese (for instance) to the old GL.

The OpenGL does not contain input code but leaves that to the operating environment, which in IRIX means X.

In short, support for internationalized input means a departure from *qread()*. Under IRIX, that means using mixed-model input, all the more reason to use a toolkit.

About X Keyboard Support

This section provides some background that may help make the following sections easier to understand.

Keys, Keycodes, and Keysyms

When a client connects to the X server, the server announces its range of *keycodes* and exports a table of *keysyms*. Each key event the client receives has a single byte *keycode*, which directly represents a physical key, and a single byte *state*, which represents currently engaged modifier keys, such as Shift or Alt.

Note: The mapping of state bits to modifiers is done by another table acquired from the server.

Keysyms are well defined, and there has been an attempt to have a keysym for every engraving one might possibly find on any keyboard, anywhere. (An *engraving* is the image imprinted on a physical key.) These are contained in */usr/include/X11/keysymdef.h*. Keysyms represent the engravings on the actual keys, but not their meanings. The server's idea of the keysym table can be changed by clients, and clients may receive *KeyMap* events when this remapping happens, but such events don't happen often.

When a client receives a Key event, it asks Xlib to use the keycode to index into its keysym table to find a list of keysyms. (This list is usually very short. Most keys have only one or two engravings on them.) Using the state byte, Xlib chooses a keysym from the list to find out what was engraved on the key the user pressed.

At this point, the client can choose to act on the keysym itself (if, for instance, it was a backspace) or it can ask for a character string represented by the keysym (or both). Generating such a string is tricky; it is discussed in "Input Methods (IMs)," below.

Details on X keyboard support can be found in *X Window System, Third Edition*, from Digital Press. Details on input methods are also available in that book, as well as in the *Xlib Programming Manual, Volume One*.

Composed Characters

There are two ways to compose characters that do not exist on a keyboard: explicit and implicit. It is common for an application to be modal and switch between the two. For example, Japanese input of *kana* is often done via implicit composition.

Users switch between a mode where input is interpreted as *romaji* (Latin characters) and a mode where strokes are all translated to *kana*.

Furthermore, both styles may operate simultaneously. While an application is supporting implicit composition of certain characters, other characters may be composable via explicit composition.

Not every keystroke produces a character, even if the associated keysym normally implies character text. The event-to-string translation routines (see "XLookupString(), XwcLookupString(), and XmbLookupString()" in this section) figure out what result a given set of keystrokes should produce.

Explicit Composition
: Explicit composition is requested when the user presses the Compose key and then types a key sequence that corresponds to the desired character. For example, to compose the character 'ñ' under some keymaps, you might press the Compose key and then type "~n".

  Note: The Compose key can be defined by using *xmodmap*(1) to map the XK_Multi_key keysym onto whatever key you want to use as Compose.

Implicit Composition
: Implicit composition mimics many existing European typewriters that have "dead" keys: keys that type a character but do not advance the carriage. When a special "dead" key is struck, the system attempts to compose a character using the next character struck. For example, on a keyboard that had a diaeresis (¨) and an O, but no Ö, one would simply strike '¨' and then 'O' to compose 'Ö'.

  Implicit composition support usually comes with some specified way to leave characters uncomposed.

Supported Keyboards

IRIX currently supports 12 keyboard layouts: American, Belgian, Danish, English, French, German, Italian, Norwegian, Portuguese, Spanish, Swedish, and Swiss. All are representable in Latin 1; the American keyboard needs only ASCII.

Input Methods (IMs)

Input methods (IMs) are ways to translate keyboard-input events into text strings. You would use a different input method, for instance, to type on a USA keyboard in Chinese than to type on the same keyboard in English. Nobody would build a keyboard suitable for direct input of the roughly 80,000 distinct Chinese characters.

IMs come in two flavors, *front-end* and *back-end*. Both types can use identical application programming interfaces, so we lose no generality by using back-end methods for our examples here.

To use an IM, follow these steps:

1. Open the IM.
2. Find out what the IM can do.
3. Agree upon capabilities to use.
4. Create input contexts with preferences and window(s) specified (see “Input Contexts (ICs)” on page 189).
5. Set the input context focus.
6. Process events.

Although all applications go through the same setup when establishing input methods, the results can vary widely. In a Japanese locale, one might end up with networked communications with an input method server and a *kanji* translation server, with circuitous paths for Key events. But in (say) a Swiss locale, it is likely that nothing would occur besides a flag or two being set in Xlib. Since operating in non-Asian locales ends up bypassing almost all of the things that might make input methods expensive, Western users are not noticeably penalized for using Asia-ready applications.

Opening an Input Method

XOpenIM() opens an input method appropriate for the locale and modifiers in effect when it is called. The locale is bound to that IM and cannot be changed. (But you could open another IM if you wanted to switch later.) Strings returned by *XmbLookupString()* and *XwcLookupString()* are encoded in the locale that was current when the IM was opened, regardless of current input context.

The syntax is:

```
XIM XOpenIM(Display *dpy, XrmDataBase db, char *res_name,
            char *res_class);
```

The *res_name* is the resource name of the application, *res_class* is the resource class, and *db* is the resource database that the input method should use for looking up resources private to itself. Any of these can be NULL.

So opening an IM is easy. For example:

Example 4-3 Opening an IM

```
XIM im;
im = XOpenIM(dpy, NULL, NULL, NULL);
if (im == NULL)
    exit_with_error();
```

XOpenIM() finds the IM appropriate for the current locale. If *XSupportsLocale()* has returned good status (see “Initialization for Xlib Programming”) and *XOpenIM()* fails, something is amiss with the administration of the system.

XSetLocaleModifiers() determines configure locale modifiers. The local host X locale modifiers announcer - XMODIFIERS environment variable is appended to the modifier list to provide default values on the locale host. The modifier list argument is a null-terminated string of the form:

“{@*category*=*value*}”

For example, if you want to connect Input Method Server - “*xwnmo*,” set modifiers “*_XWNMO*” as follows:

- `XSetLocaleModifiers(“@im=_XWNMO”);`

or

- set environment variable “`XMODIFIERS=@im=_XWNMO`” and `XSetLocaleModifiers(“”);`.

Note: The library routines are not prepared for the possibility of XSupportsLocale() succeeding and XOpenIM() failing, so it’s up to application developers to deal with such an eventuality. (This circumstance

could occur, for example, if the IM died after XSupportsLocale() was called.) This topic is under some debate in the MIT X consortium. If XSetLocaleModifiers() is wrong, XOpenIM() will fail.

Most of the complexity associated with IM use comes from configuring an input context to work with the IM. Input contexts are discussed in "Input Contexts (ICs)" on page 189.

To close an input method, call *XCloseIM()*.

IM Styles

If the application requests it, an input method can often supply *status* information about itself. For example, a Japanese IM may be able to indicate whether it is in Japanese input mode or *romaji* input mode. An input method can also supply *pre-edit* information, partial feedback about characters in the process of being composed. The way in which an IM deals with status and pre-edit information is referred to as an IM *style*. This section describes styles and their naming.

Root Window

The *Root Window* style has a pre-edit area and a status area in a window owned by the IM as a descendant of the root. The application does not manage the pre-edit data, the pre-edit area, the status data, or the status area. Everything is left to the input method to do in its own window, as illustrated in Figure 4-1:

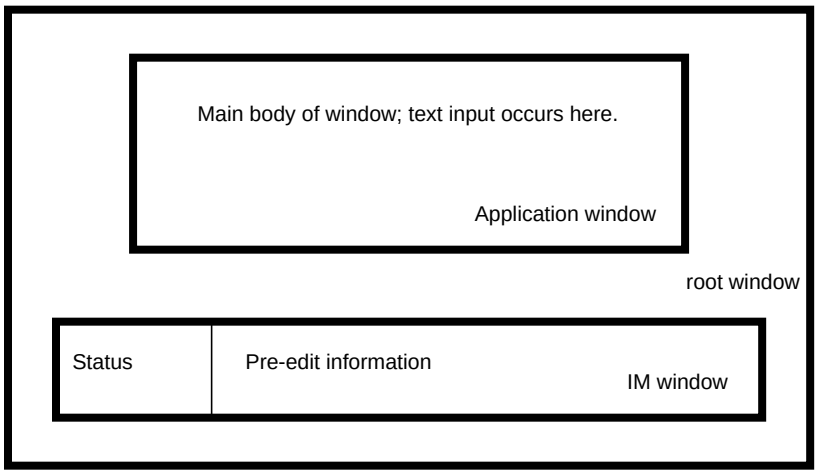

Figure 4-1 Root Window Input

Off-the-Spot

The *Off-the-Spot* style places a pre-edit area and a status area in the window being used, usually in reserved space away from the place where input appears. The application manages the pre-edit area and status area, but allows the IM to update the data there. (The application provides information regarding foreground and background colors, fonts, and so on.) A window using Off-the-Spot input style might look like that shown in Figure 4-2:

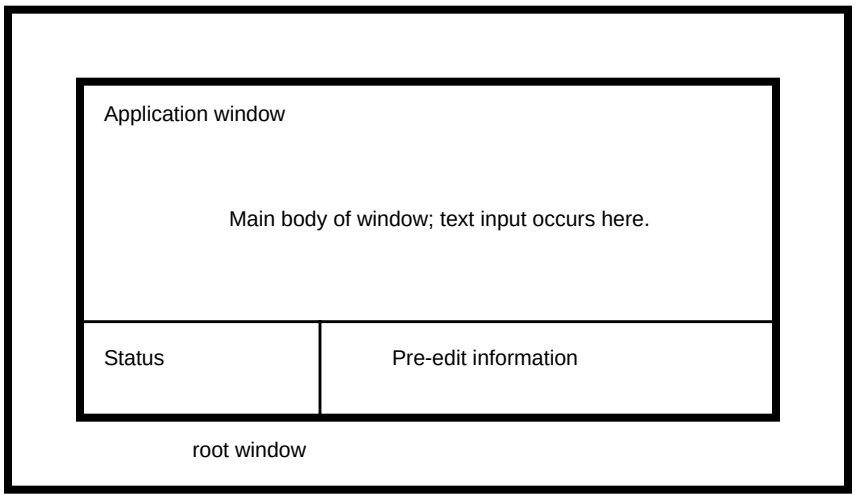

Figure 4-2 Off-the-Spot Input

Over-the-Spot

The *Over-the-Spot* style involves the IM creating a small, pre-edit window over the point of insertion. The window is owned and managed by the IM as a descendant of the root, but it gives the user the impression that input is being entered in the right place; in fact, the pre-edit window often has no borders and is invisible to the user, giving the appearance of On-the-Spot input. The application manages the status area as in Off-the-Spot, but specifies the location of the editing so that the IM can place pre-edit data over that spot.

On-the-Spot

On-the-Spot input is by far the most complex for the application developer. The IM delivers all pre-edit data via callbacks to the application, which must perform in-place editing—complete with insertion and deletion and so on. This approach usually involves a great deal of string and text rendering support at the input generation level, above and beyond the effort required for completed input. Since this may mean a lot of updating of surrounding data or other display management, everything is left to the application. There is little chance an IM could ever know enough about the application

to be able to help it provide user feedback. The IM therefore provides status and edit information via callbacks.

Done well, this style can be the most intuitive one for a user.

Setting IM Styles

A style describes how an IM presents its pre-edit and status information to the user. An IM supplies information detailing its presentation capabilities. The information comes in the form of flags, OR'ed together. The flags to use with each style are:

Root Window	XIMPreeditNothing | XIMStatusNothing
Off-the-Spot	XIMPreeditArea | XIMStatusArea
Over-the-Spot	XIMPreeditPosition | XIMStatusArea
On-the-Spot	XIMPreeditCallbacks | XIMStatusCallbacks

For example, if you wanted a style variable to match an Over-the-Spot IM style, you could write:

```
XIMStyle over = XIMPreeditPosition | XIMStatusArea;
```

If an IM returns *XIMStatusNone* (not to be confused with *XIMStatusNothing*), it means the IM will not supply status information.

Using Styles

An input method supports one or more styles. It's up to the application to find a style that is supported by both the IM and the application. If several exist, the application must choose. If none exist, the application is in trouble.

Input Contexts (ICs)

An input method may be serving multiple clients, or one client with multiple windows, or one client with multiple input styles on one window. The specification of style and client/IM communication is done via *input contexts*. An input context is simply a collection of parameters that together describe how to go about receiving and examining input under a given set of circumstances.

To set up and use an input context:

1. Decide what styles your application can support.
2. Query the IM to find out what styles it supports.
3. Find a match.
4. Determine information that the IC needs in order to work with your application.
5. Create the IC.
6. Employ the IC.

Find an IM Style

The IM may be able to support multiple styles—for example, both Off-the-Spot and Root Window. The application may be able to do, in order of preference, Over-the-Spot, Off-the-Spot, and Root Window. The application should determine that the best match in this case is Off-the-Spot.

First, discover what the IM can do, then set up a variable describing what the application can do:

```
XIMStyles *IMcando;
XIMStyle  clientCanDo; /* note type difference */
XIMStyle  styleWeWillUse = NULL;

XGetImValues(im, XNQueryInputStyle, &IMcando, NULL);

clientCanDo =
/*none*/ XIMPreeditNone | XIMStatusNone |
/*over*/ XIMPreeditPosition | XIMStatusArea |
/*off*/  XIMPreeditArea | XIMStatusArea |
/*root*/ XIMPreeditNothing | XIMStatusNothing;
```

A client should always be able to handle the case of *XIMPreeditNone* | *XIMStatusNone*, which is likely in a Western locale. To the application, this is not very different from a *RootWindow* style, but it comes with less overhead.

Once we know what the application can handle, we look through the IM styles for a match:

```
for(i=0; i<IMcando->count_styles; i++) {
    XIMStyle tmpStyle;
```

```
    tmpStyle = IMcando->support_styles[i];

    if ( ((tmpStyle & clientCanDo) == tmpStyle) &&
        prefer(tmpStyle, styleWeWillUse) )
            styleWeWillUse = tmpStyle;
}

if (styleWeWillUse = NULL)
    exit_with_error();
XFree(IMcando);

/* styleWeWillUse is set, which is what we were after */
```

The *prefer()* routine simply applies some heuristic to application preference.

IC Values

There are several pieces of information an input method may require, depending on the input context and style chosen by the application. The input method can acquire any such information it needs from the input context, ignoring any information that does not affect the style or IM.

A full description of every item of information available to the IM is supplied in *X Window System, Third Edition*. We include only a brief list here:

- *XNClientWindow* specifies to the IM which client window it can display data in or create child windows in. It is set once and cannot be changed.
- *XNFilterEvents* is an additional event mask for event selection on the client window.
- *XNFocusWindow* specifies the focus window. This specifies which window gets the processed (composed) Key events.
- *XNGeometryCallback* specifies a geometry handler that is called if the client allows an IM to change the geometry of the window.
- *XNInputStyle* specifies the style for this IC.
- *XNResourceClass* and *XNResourceName* specify the resource class and name to use when the IM looks up resources that vary by IC.
- *XNStatusAttributes* and *XNPreeditAttributes* specify to an IM the attributes to be used for any status and pre-edit areas. The attributes are nested, variable-length lists.

Pre-edit and Status Attributes

When an IM is going to provide state, it needs some simple X information with which to do its work. For example, if an IM is going to draw status information in a client window in an Off-the-Spot style, it needs to know where the area is, what color and font to render text in, and so on. The application gives this data to the IC for use by the IM.

As with the "IC Values" section, full details are available in *X Window System, Third Edition*.

- *XNArea* specifies a rectangle to be used as a status or pre-edit area.
- *XNAreaNeeded* specifies the rectangle desired by the attribute writer. Either the application or the IM may provide this information, depending on circumstances.
- *XNBackgroundPixmap* specifies a pixmap to be used for the background of windows the IM creates.
- *XNColormap* specifies the colormap to use.
- *XNCursor* specifies the cursor to use.
- *XNFontSet* specifies the fontset to use for rendering text.
- *XNForeground* and *XNBackground* specify the colors to use for rendering.
- *XNLineSpacing* specifies the line spacing to be used in the pre-edit window if more than one line is used.
- *XNSpotLocation* specifies where the next insertion point is, for use by *XIMPreeditPosition* styles.
- *XNStdColormap* specifies that the IM should use *XGetRGBColormaps()* with the supplied property (passed as an Atom) in order to find out which colormap to use.

Creating an Input Context

Creating an input context is a simple matter of calling *XCreateIC()* with a variable-length list of parameters specifying IC values. Here's a simple example that works for the root window. An example follows.

Example 4-4 Creating an Input Context with ***XCreateIC()***

```
XVaNestedList arglist;
XIC ic;

arglist = XVaCreateNestedList(0, XNFontSet, fontset,
                              XNForeground,
                              WhitePixel(dpy, screen),
                              XNBackground,
                              BlackPixel(dpy, screen),
                              NULL);

ic = XCreateIC(im, XNInputStyle, styleWeWillUse,
               XNClientWindow, window, XNFocusWindow, window,
               XNStatusAttributes, arglist,
               XNPreeditAttributes, arglist, NULL);
XFree(arglist);

if (ic == NULL)
    exit_with_error();
```

Using the IC

A multi-window application may choose to use several input contexts. But for simplicity, we assume that the application just wants to get to the internationalized input using one method in one window.

Using the IC is a matter of (a) making sure we check events the IC wants and (b) setting IC focus. If you are setting up a window for the first time, you know the event mask you want, and you can use it directly. If you are attaching an IC to a previously configured window, you should query the window and add in the new event mask.

Example 4-5 Using the IC

```
unsigned long imEventMask;

XGetWindowAttributes(dpy, win, &winAtts);
XGetICValues(ic, XNFilterEvents, &imEventMask, NULL);
```

```
imEventMask |= winAtts.your_event_mask;
XSelectInput(dpy, window, imEventMask);
XSetICFocus(ic);
```

At this point, the window is ready to be used.

Events Under IM Control

Processing events under input method control is almost the same in X11R6 as it was under R4 and before. There are two essential differences: *XFilterEvent()* and the *LookupString* routines.

XFilterEvent()

Every event received by your application should be fed to the IM via *XFilterEvent()*, which returns a value telling you whether or not to disregard the event. IMs asks you to disregard the event if they have extracted the data and plan on giving it to you later, possibly in some other form. All events (not just *KeyPress* and *KeyRelease* events) go to *XFilterEvent()*.

If we compacted the event processing into a single routine, a typical event loop would look something like the following example.

Example 4-6 Event Loop

```
Xevent event;

while (TRUE) {
    XNextEvent(dpy, &event);

    if (XFilterEvent(&event, None))
        continue;

    DealWithEvent(&event);
}
```

XLookupString(), *XwcLookupString()*, and *XmbLookupString()*

When using an input method, you should replace calls to *XLookupString()* with calls to *XwcLookupString()* or *XmbLookupString()*. The *MB* and *WC* versions have very similar interfaces. In the examples below, we arbitrarily use *XmbLookupString()*, but the examples apply to both versions.

There are two new situations to deal with:

1. The string returned may be long.
2. There may be an interesting keysym returned, an interesting set of characters returned, both, or neither.

Dealing with the former is merely a matter of maintaining an arena, as in the example below.

To tell the application what to pay attention to for a given event, *XmbLookupString()* returns a status value in a passed parameter, equal to one of the following:

- *XLookupKeysym*, which indicates the keysym should be checked
- *XLookupChars*, which indicates a string has been typed or composed
- *XLookupBoth*, which means both of the above
- *XLookupNone*, which means neither is ready for processing
- *XBufferOverflow*, which means the supplied buffer is too small—*XmbLookupString()* should be called again with a bigger buffer

XmbLookupString() also returns the length of the string in question. Note that *XmbLookupString()* returns the length of the string in bytes, while *XwcLookupString()* returns the length of the string in characters.

The example below should help show how these functions work. Most event processors perform a *switch()* on the event type; assume we have done that, and we have received a *KeyPress* event.

Example 4-7 *KeyPress* Event

```
case KeyPress:
{
    Keysym keysym;
    Status status;
    int buflength;
    static int bufsize = 16;
    static char *buf = NULL;

    if (buf == NULL) {
        buf = malloc(bufsize);
        if (buf < 0) StopSequence();
```

```
    }
    buflength = XmbLookupString(ic, &event, buf, bufsize,
                                &keysym, &status);
    /* first, check to see if that worked */
    if (status == XBufferOverflow) {
        buf = realloc(buf, (bufsize = buflength));
        buflength = XmbLookupString(ic, &event, buf, bufsize,
                                    &keysym, &status);
    }
    /* We have a valid status. Check that */
    switch(status) {
    case XLookupKeysym:
    case XLookupBoth:
        DealWithKeysym(keysym);
        if (status == XLookupKeysym)
            break; /* wasn't XLookupBoth */
    case XLookupChars:
        DealWithString(buf, buflength);
    case XLookupNone:
        break;
    } /* end switch(status) */
} /* end case KeyPress segment */
break; /* we are in a switch(event.type) statement */
```

GUI Concerns

It shouldn't be significantly more difficult to internationalize an application with a graphical user interface than an application without such an interface, but there are a few further issues that must be addressed. These include:

- "X Resources for Strings" covers labeling objects using X resources.
- "Layout" describes creating layouts that are usable after localization.
- "Icons" explains some concerns for localizing icons.

X Resources for Strings

Resource lookup mechanisms in Xlib as well as in toolkits monitor locale environment variables when locating resource files. For string constants that

are used within toolkit objects, resources provide a simpler solution than do message catalogs.

Some common objects that should definitely get their labels from resources:

- Labels
- Buttons
- Menu items
- Dialog notices and questions

Any object that employs some sort of text label should be labeled via resources. Since the localizer wants to provide strings for the local version of the application, the app-defaults file for the application should specify every reasonable string resource. Reference pages should identify all localizable string resources.

Localizers of an application provide a separate resource file for each locale that the application runs in.

Layout

Layout management is of special interest when you cannot predict how large a button or other label might be. The nature of the problem of layout composition and management does not change, but one must construct the layout management without full knowledge of the final appearance.

It's worth noting that localization efforts can be assumed to be "reasonable" in some sense. For example, X resources have always allowed a user to specify an extremely large font for buttons, but applications correctly choose to let such users live with the results. But it's not always that clear what is reasonable and what isn't; we don't always know what will be difficult to translate succinctly in some locale. So while developers need not provide for all combinations of resource specifications, they must take the responsibility to make the application localizable.

Three main approaches to the layout problem include:

- Dynamic Layout
- Constant Layout
- Localized Layout

Dynamic Layout

Most toolkits provide *form*, *pane*, *rowcolumn*, *rubberboard*, or other layout objects that will calculate layout depending on the "natural" (localized) size of the objects involved. Most use some hints provided by the developer that can regulate this layout. For example, some IRIS IM widgets providing these services are *XmForm*, *XmPanedWindow*, and *XmRowColumn*.

Dynamic layout is probably the simplest way to prevent localization difficulties.

Note: The IRIS IM product is the Silicon Graphics port of the OSF / Motif product, and should not be confused with IM, the abbreviation for Input Methods.

Constant Layout

Under certain circumstances, an application may insist on having a predefined layout. When this is so, the application must provide objects that are so constructed as to allow localization. A "Quit" button that just barely allows room for the Latin 1 string "Quit" is not likely to suffice when localizers attempt to fit their translations into that small space.

In order to enforce constant layout, the developer incurs the heavy responsibility of making sure the objects are localizable. This means a lot of investigation; the "there, that ought to be enough" approach is chancy at best.

Localized Layout

Some toolkits provide for layout control by run-time reading of strings or other data files. Applications that use such toolkits can easily finesse the layout issue by providing capability for localization of the layout as well as the contents of the layout. This provides each localizer maximum freedom in

presenting the application to the local users. The application developer is responsible for providing localizers with instructions and mechanisms necessary to produce layout data.

IRIS IM Localization with *editres*

IRIX provides an interactive method of laying out widgets for IRIS IM and Xaw (the Athena Widget Set): a utility called *editres*. With *editres*, you can construct and edit resources and see how your widgets will look on the screen; the program even generates a usable app-defaults file for you. But note that if you hard-code any resources into your IRIS IM code, you won't be able to edit them using this method.

Icons

Icons attempt to be fairly generic representations of their antecedents. Unfortunately, it is very difficult for a designer to know what is generic or recognizable in other cultures. Therefore, it is important that any pictographic representations used by an application be localizable.

Graphic representations can be stored as strings representing X bitmaps, as names of data files containing pictographs, or in whatever manner the developer thinks best, so long as the developer provides a way for the localizer to produce and deliver localized pictographs.

Popular Encodings

This section discusses three encodings that are commonly used:

- "The ISO 8859 Family" explains the ISO 8859 family of encodings.
- "Asian Languages" describes Asian language encodings.
- "ISO 10646 and Unicode" covers the ISO 10646 and Unicode.

The ISO 8859 Family

American English is easily representable in 7-bit ASCII. Most other languages are not. For example, the character é is not in ASCII.

Most Western European languages are representable in 8-bit ISO 8859-1, which is commonly known as Latin 1. Latin 1 is a superset of ASCII including characters used by several Western European languages (such as ö, £, ñ, ç, ¿).

ISO 8859 comes in nine parts, many of which overlap; all are supersets of ASCII.

The ISO 8859 Character Sets are shown in Table 4-4.

Table 4-4 ISO 8859 Character Sets

Character set	Common name	Languages supported
8859-1	Latin 1	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish
8859-2	Latin 2	Albanian, Czech, English, German, Hungarian, Polish, Rumanian, Serbo-Croatian, Slovak, Slovene
8859-3	Latin 3	Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish
8859-4	Latin 4	Danish, English, Estonian, Finnish, German, Greenlandic, Lapp, Latvian, Lithuanian, Norwegian, Swedish
8859-5	Latin / Cyrillic	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo-Croatian, Ukrainian
8859-6	Latin / Arabic	Arabic, English (see ISO 8859-6 specification)
8859-7	Latin / Greek	English, Greek (see ISO 8859-7 specification)

Table 4-4 (continued) ISO 8859 Character Sets

Character set	Common name	Languages supported
8859-8	Latin / Hebrew	English, Hebrew (see ISO 8859-8 specification)
8859-9	Latin 5	Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish

IRIX contains over 500 Latin 1 fonts, as well as a few fonts for each of the other 8859-encoded character sets except 8859-6. Currently, IRIX contains no fonts for use with the 8859-6 character set.

To get the list of ISO-8859 fonts, enter the following:

```
xlsfonts
```

Or you can restrict the amount of output, for example, by typing:

```
xlsfonts '*8859-2'
```

To see the encoding, use the *xfd* command. For example:

```
xfd -fn -sgi-screen-medium-r-normal--9-90-72-72-m-60-iso8859-1
```

For more information on *xlsfonts* and *xfd*, and installing and using fonts, refer to Chapter 3, “Working With Fonts.”

Asian Languages

Asian languages are commonly pictographic and employ many thousands of characters for their representation. For example, Japanese and Korean can be practically encoded in 16 bits. Daily-use Chinese can also be, but archives and scholars frequently need more; Chinese is often encoded with up to four bytes per character.

Some Standards

Various Asian character sets have been developed, some of which are considered standard. Encodings for these sets are less standardized. Asian

character sets usually require larger-than-byte character types like those described in “Multibyte Characters.” Table 4-5 lists some of these standard character sets. Note that some of these character sets have multiple associated codesets, usually designated by appending the year the codeset was adopted to the character set name. (For example, JIS X 208-1983 is different from JIS X 208-1990.)

Table 4-5 Character Sets for Asian Languages

Language	Character Set Standards	Support
Japanese	JIS X 0201.1976-0	*Katakana*
	JIS X 0208.1983-0	*Kanji*, *kana*, Latin, Greek, Cyrillic, symbols, others
	JIS X 0212.1990-0	Supplemental *kanji*, others
Chinese	GB2312.1980-0	
Korean	KSC5601.1987-0	Hangul
Taiwan	CNS11643	

EUC

EUC is *Extended Unix Code*, an encoding methodology that supports concurrent use of four codesets in one encoding. It employs two special “shift state” bytes:

```
ss1 = 0x8e
ss2 = 0x8f
```

These are used to identify codesets within a string. The EUC encoding scheme uses the following patterns to indicate which codeset is in use at any given time:

```
Codeset #1: 0xxxxxxx
Codeset #2: 1xxxxxxx [ 1xxxxxxx ...]
Codeset #3: ss1 1xxxxxxx [ 1xxxxxxx ...]
Codeset #4: ss2 1xxxxxxx [ 1xxxxxxx ...]
```

So if *ss1* appears in a string, it means that the next character—however many bytes long it is—should be interpreted as a character from codeset #3. (So if there are multiple characters in a row from codeset #3, each one is preceded by *ss1*.) Similarly, *ss2* indicates that the following character belongs to codeset #4. If any other byte whose high bit is 1 appears in the string (without being preceded by *ss1* or *ss2*), it is interpreted as part of a character from codeset #2.

In EUC, codeset #1 is always ASCII. The other codesets are implementation- or user-defined.

EUC implementations exist (but are not standardized) for all ideographic Asian languages.

ISO 10646 and Unicode

ISO and the Unicode Consortium have jointly developed a character set designed to cover almost every character normally used by any language in the world. ISO calls this *ISO IS 10646*. The Unicode Consortium embraces a subset of 10646, called the *Basic Multilingual Plane* (BMP) of 10646, and calls it *Unicode*. The only characters defined in either standard are the characters in the BMP. The characters have 2- and 4-byte representations.

It appears that ISO 10646 will grow significantly in acceptance, but widespread use is still some years away.

Appendix A

ISO 3166 Country Names and Abbreviations

This appendix alphabetically lists ISO 3166 country codes.

Appendix A

ISO 3166 Country Names and Abbreviations

Table A-1 lists the ISO 3166 country codes, alphabetized by country name (the table reads from left to right, and top to bottom).

Table A-1 ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Afghanistan	AF	Albania	AL	Algeria	DZ
American Samoa	AS	Andorra	AD	Angola	AO
Anguilla	AI	Antarctica	AQ	Antigua and Barbuda	AG
Argentina	AR	Aruba	AW	Australia	AU
Austria	AT	Bahamas	BS	Bahrain	BH
Bangladesh	BD	Barbados	BB	Belgium	BE
Belize	BZ	Benin	BJ	Bermuda	BM
Bhutan	BT	Bolivia	BO	Botswana	BW
Bouvet Island	BV	Brazil	BR	British Indian Ocean Territory	IO
Brunei Darussalam	BN	Bulgaria	BG	Burkina Faso	BF
Burma	BU	Burundi	BI	Byelorussia	BY
Cameroon	CM	Canada	CA	Cape Verde	CV
Cayman Islands	KY	Central African Republic	CF	Chad	TD
Chile	CL	China	CN	Christmas Island	CX
Cocos Islands	CC	Colombia	CO	Comoros	KM

Table A-1 (continued) ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Congo	CG	Cook Islands	CK	Costa Rica	CR
Cote D'Ivoire	CI	Cuba	CU	Cyprus	CY
Czech Republic	CS	Denmark	DK	Djibouti	DJ
Dominica	DM	Dominican Republic	DO	East Timor	TP
Ecuador	EC	Egypt	EG	El Salvador	SV
Equatorial Guinea	GQ	Ethiopia	ET	Falkland Islands	FK
Faroe Islands	FO	Fiji	FJ	Finland	FI
France	FR	French Guiana	GF	French Polynesia	PF
French Southern Territories	TF	Gabon	GA	Gambia	GM
Germany	DE	Ghana	GH	Gibraltar	GI
Greece	GR	Greenland	GL	Grenada	GD
Guadalupe	GP	Guam	GU	Guatemala	GT
Guinea-Bissau	GW	Guinea	GN	Guyana	GY
Haiti	HT	Heard and McDonald Islands	HM	Honduras	HN
Hong Kong	HK	Hungary	HU	Iceland	IS
India	IN	Indonesia	ID	Iran	IR
Iraq	IQ	Ireland	IE	Israel	IL
Italy	IT	Jamaica	JM	Japan	JP
Jordan	JO	Kampuchea	KH	Kenya	KE
Kiribati	KI	Korea	KP or KR	Kuwait	KW

Table A-1 (continued) ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Laos	LA	Lebanon	LB	Lesotho	LS
Liberia	LR	Libya	LY	Liechtenstein	LI
Luxembourg	LU	Macau	MO	Madagascar	MG
Malawi	MW	Malaysia	MY	Maldives	MV
Mali	ML	Malta	MT	Marshall Islands	MH
Martinique	MQ	Mauritania	MR	Mauritius	MU
Mexico	MX	Micronesia	FM	Monaco	MC
Mongolia	MN	Montserrat	MS	Morocco	MA
Mozambique	MZ	Namibia	NA	Nauru	NR
Nepal	NP	Netherlands Antilles	AN	Netherlands	NL
Neutral Zone	NT	New Caledonia	NC	New Zealand	NZ
Nicaragua	NI	Nigeria	NG	Niger	NE
Niue	NU	Norfolk Island	NF	Northern Mariana Islands	MP
Norway	NO	Oman	OM	Pakistan	PK
Palau	PW	Panama	PA	Pangaea	GE
Papua New Guinea	PG	Paraguay	PY	Peru	PE
Philippines	PH	Pitcairn	PN	Poland	PL
Portugal	PT	Puerto Rico	PR	Qatar	QA
Quebec	QC	Reunion	RE	Romania	RO
Rwanda	RW	Saint Kitts and Nevis	KN	Saint Lucia	LC
Saint Vincent and the Granadines	VC	Samoa	WS	San Marino	SM

Table A-1 (continued) ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Sao Tome and Principe	ST	Saudi Arabia	SA	Senegal	SN
Seychelles	SC	Sierra Leone	SL	Singapore	SG
Solomon Islands	SB	Somalia	SO	South Africa	ZA
Spain	ES	Sri Lanka	LK	St. Helena	SH
St. Pierre and Miquelon	PM	Sudan	SD	Suriname	SR
Svalbard and Jan Mayen Islands	SJ	Swaziland	SZ	Sweden	SE
Switzerland	CH	Syrian Arab Republic	SY	Taiwan	TW
Tanzania	TZ	Thailand	TH	Togo	TG
Tokelau	TK	Tonga	TO	Trinidad and Tobago	TT
Tunisia	TN	Turkey	TR	Turks and Caicos Islands	TC
Tuvalu	TV	Uganda	UG	Ukraine	UA
United Arab Emirates	AE	United Kingdom	GB	United States Minor Outlying Islands	UM
Uruguay	UY	Vanuatu	VU	Vatican City State	VA
Venezuela	VE	Viet Nam	VN	Virgin Islands (British)	VG
Virgin Islands (USA)	VI	Wallis and Futuna Islands	WF	Western Sahara	EH
Yemen	YE or YD	Yugoslavia (Former)	YU	Zaire	ZR
Zambia	ZM	Zimbabwe	ZW		

Index

F

G

H

I

M

N

U

V

W

X

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2478-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  1600 Amphitheatre Pkwy, M/S 535
  Mountain View, California 94043-1351