FUJITSU COMPUTERS SIEMENS

Reliant UNIX ONLINE Documentation

Reliant UNIX 5.43

Integrated Software Development Guide

Edition March 1997

Identification: U6397-J-Z145-3-7600

Copyright and Trademarks

1 Introduction

Synopsis

This manual describes the commands, functions and software products (programs) associated with

- application programming in the Reliant UNIX system environment, and
- application and driver software packaging

Target groups

This manual is intended for application software programmers and for users responsible for packaging application and driver software.

Summary of contents

The opening chapter of this manual starts with a general introduction to the subject of application programming, referring you to tools you can use for application development. Then you are introduced to various aspects of application software development. Thereafter the remainder of the manual is divided into three parts.

The first part covers aspects of application programming in a Reliant UNIX system environment. Here you will find information on topics such as interprocess communication, memory management and symbolic links.

The second part deals with aspects of application and driver software packaging. Here you can find out about data validation tools, device drivers and other similar topics.

The third part consists of descriptions of commands and functions required for application programming and packaging.

1.1 About this manual

This manual is the result of merging the documents PGSSAPT and ISDGD Version 3 (USL) to form ISDGD Version 4 (PGSSAPT = Programmer's Guide: System Services and Application Packaging Tool, ISDGD = Integrated Software Development Guide). The manual is in three main parts covering:

1. application programming in the Reliant UNIX system environment
2. application and driver software packaging
3. reference pages (man pages) dealing with application programming and packaging

Thus it supersedes the manuals "Integrated Software Development Guide (ISDG)" [12] and "System Services and Application Packaging Tools" [2].

In this manual you will find numerous references to other manuals, especially to the "Programmer's Guide: ANSI C and Programming Support Tools" [3]. The "Programmer's Guide: ANSI C and Programming Support Tools" [3] and the manual "System Services and Application Packaging Tools" [2] are closely connected, so you will also find a number of references to the latter manual.

The "Programmer's Guide: ANSI C and Programming Support Tools" [3] describes the C programming environment, libraries, compiler, link editor, and file formats. It also describes the tools provided in the Reliant UNIX System/C environment for building, analyzing, debugging, and maintaining programs.

The manual "System Services and Application Packaging Tools" [2] concentrates on an application programmer's view of how to develop and package application software under Reliant UNIX V, using the services provided by the system.

If you are unsure of which book to reference, check the "Product Overview" [1]. It explains how the document set is organized and where to find specific information.

1.1.1 Machines and machine dependencies

This manual contains a wealth of information and examples relating to application programming and packaging. In order neither to restrict the range of information supplied nor to make the book unnecessarily large, on the whole no attempt has been made to draw hardware-based distinctions, for example between the MX300, MX500 and the WX200. Thus it is possible that some descriptions, functions and examples will not

apply to all machines.

1.1.2 Configuration and configuration dependencies

The way in which functions, commands and examples operate may depend on how your system is configured. This typically relates to default directories and preset environment variables. This manual makes no reference to configuration dependencies of this type.

1.2 Introduction to application programming

This section introduces application programming in a Reliant UNIX system environment.

It first describes what application programming is and then moves on to a discussion on tools and languages and where you can read about them.

In the Reliant UNIX world the term application programming refers to development using the services of system calls and library functions (system programming is the term used for development of parts of the Reliant UNIX operating system).

The following general observations apply to programming:

- Large programs are developed by a team of people who write requirements, designs, tests, and end-user documents. This implies use of a project management methodology, including version control (described in [3]), change requests, tracking, and so on.
- Programs must be developed robustly.

  They should be easy to use, implying character or graphical user interfaces.

  They should check all incoming data for validity (for example, using the data validation tools described in the corresponding section of this manual).

  They should be able to handle large amounts of data.
- Programs must be easy to install and administer (see Section "Packaging application software", and Section "Modifying the sysadm interface".)

1.2.1 Reliant UNIX system tools and where you can read about them

The term "Reliant UNIX system tools" refers to the services supplied by the system to help developers solve programming problems. This includes both the command level and programming languages.

1.2.2 Tools covered and not covered in this guide

This manual deals with tools used in the process of creating programs in a Reliant UNIX system environment.

Tools that are covered here apply to applications. Each application performs a different function, but goes through the same basic steps: input, processing, and output. These tools help you accomplish these steps.

The manual also deals with tools for packaging applications software and customizing the user interface. The following section introduces a number of commands and languages supported in a Reliant UNIX environment.

The following topics are not covered in this guide:

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

Information about these subjects can be found in the "User's Guide" [11] and a number of commercially available texts.

1.2.3 Supported languages and language-like components

By "languages" we mean those running on a Reliant UNIX system. Since these are separately purchasable items, not all of them will necessarily be installed on your machine. On the other hand, you may have languages available on your machine that came from another source and are not mentioned in this discussion.

1.2.3.1 The C language

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

- a variety of data types: characters, integers of various sizes, and floating point numbers
- low-level functions (most of the Reliant UNIX system kernel is written in C)
- derived data types such as arrays, functions, pointers, structures, and unions
- multidimensional arrays
- scaled pointers and the ability to do pointer arithmetic
- bitwise operators
- a variety of flow-of-control statements: if, if-else, switch, while, do-while, and for
- a high degree of portability

Refer to [3] for full details of C.

1.2.3.2 Shell

You can use the shell to create programs (new commands). Such programs are also called shell procedures. Refer to the "User's Guide" [11] for information on how to create and execute shell programs using commands, variables, positional parameters, return codes, and basic programming control structures.

1.2.3.3 awk

awk (its name is an acronym constructed from the initials of its developers) scans an input file for lines that match pattern(s) described in a specification file. On finding a line that matches a pattern, *awk* performs actions also described in the specification. It is not uncommon that an *awk* program can be written in a couple of lines to do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a quantity, and the task is to output the sum of the quantities for each key. The pseudocode for such a program might look like this:

```
SORT□RECORDS
Read□the□first□record□into□a□hold□area;
Until□EOF:
{
If□the□key□matches□the□key□of□the□record□in□the□hold□area,
--add□the□quantity□to□the□quantity□field□of□the□held□record;
If□the□key□does□not□match□the□key□of□the□held□record,
-discard□the□held□record,
--move□the□new□record□to□the□hold□area;
}
```

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

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

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

For detailed information on *awk*, see *awk* in [11] and *awk(1)* in [4].

1.2.3.4 lex

lex is a lexical analyzer which can be added to C or FORTRAN programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. *lex* can produce C language subroutines that recognize regular expressions specified by the user,

take some action when a regular expression is recognized, and pass the output stream on to the next program.

For detailed information on *lex* see the *lex* chapter in [3] and *lex(1)* in [9].

1.2.3.5 yacc

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

For detailed information on *yacc* see the *yacc* chapter in [3] and *yacc*(1) in [9].

1.2.3.6 m4

m4 is a macro processor that can be used as a preprocessor for assembly language and C programs. For details, see the *m4* chapter in [3] and *m4*(1) in [9].

1.2.3.7 bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can enter formulae in a file or at the keyboard, for example, and call *bc* to execute them. The *bc* program uses *dc*. You can use *dc* directly, if you want, but it takes a little getting used to since it works with reverse Polish notation. That means that you enter the numbers first, followed by the operator. *bc* and *dc* are described in [4].

1.2.3.8 curses

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

For detailed information on *curses*, see the [15].

1.2.3.9 FMLI

The Form and Menu Language Interpreter (FMLI) is a high-level programming tool having two main parts:

- The Form and Menu Language, a programming language for writing scripts that define how an application will be presented to users. The syntax of the Form and Menu Language is very similar to that of the Reliant UNIX system shell programming language, including variable setting and evaluation, built-in commands and functions, use of and escape from special characters, redirection of input and output, conditional statements, interrupt signal handling, and the ability to set various terminal attributes. The Form and Menu Language also includes sets of descriptors, which are used to define or customize attributes of frames and other objects in your application.
- The Form and Menu Language Interpreter *fmli*, a command interpreter that sets up and controls the video display screen on a terminal. FMLI scripts can also invoke Reliant UNIX system commands and C executables, either in the background or in full screen mode. The Form and Menu Language Interpreter operates similarly to the Reliant UNIX command interpreter *sh*. At run time it parses the scripts you have written, thus giving you the advantages of quick prototyping and easy maintenance.

FMLI provides a framework for developers to write applications and application interfaces that use menus and forms. It controls many aspects of screen management for you. That means that you do not have to be concerned with the low-level details: creating or placing frames, providing users with a means of navigating between or within frames, or processing the use of forms and menus. Nor do you need to worry about which kind of terminal an application will be run on. FMLI takes care of all that for you.

For details see the FMLI chapter in [15].

1.2.3.10 ETI

The Extended Terminal Interface (ETI) is a set of C library routines that promote the development of application programs displaying and manipulating windows, panels, menus, and forms and that run under the Reliant UNIX system.

ETI consists of

- the low-level (*curses*) library
- the *panel* library
- the *menu* library
- the *form* library
- the TAM Transition library

The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine printw that behaves much like printf and another routine getch that behaves like getc. The automatic teller program at your bank might use printw to print its menus and getch to accept your requests for withdrawals (or, better yet, deposits). Visual screen editors might also use these and other ETI routines.

A major feature of ETI is cursor optimization. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with ETI routines and edited the sentence

ETI is a great package for creating forms and menus.

to read

ETI is the best package for creating forms and menus.

the program would change only "the best" in place of "a great." The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which an ETI program is run. This means that ETI can do whatever is required to update many different terminal types. It searches the terminfo database to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your Reliant UNIX system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple ETI program. It uses some of the basic ETI routines to move a cursor to the middle of a terminal screen and print the character string BullsEye. For now, just look at their names and you will get an idea of what each of them does:

```
#include□<curses.h>
main()
□{
□□□□initscr();
□□□move(□LINES/2□-□1,□COLS/2□-□4□):
□□□addstr("Bulls");
□□□□refresh();
□□□□addstr("Eye");
□□□□refresh();
□□□□endwin();
□}
```

For complete information on ETI, refer to the ETI chapter in [15].

1.2.3.11 XWIN Graphical Windowing System

The XWIN Graphical Windowing System is a network-transparent window system. X display servers run on computers with either monochrome or color bitmap display hardware. The server distributes user input to and accepts output requests from various application programs (referred to as "clients"). Each client is located on either the same machine or on another machine in the network.

The clients use Xlib, a C library routine, to interface with the window system by means of a stream connection.

"Widgets" are a set of code and data that provide the look and feel of a user interface. The C library routines

used for creating and managing widgets are called the X Intrinsics. They are built on top of the X Window System, monitor events related to user interactions, and dispatch the correct widget code to handle the display. Widgets can then call application-registered routines (called callbacks) to handle the specific application semantics of an interaction. The X Intrinsics also monitor application-registered, nongraphical events and dispatch application routines to handle them. These features allow programmers to use this implementation of a Reliant UNIX/windows toolkit in data base management, network management, process control, and other applications requiring response to external events.

Clients sometimes use a higher level library of the X Intrinsics and a set of widgets in addition to xlib. Refer to "XWIN Graphical Windowing System" [13] for general information about the design of X. The "Xlib\C Language Interface" manual [14] is a reference guide to the low-level C language interface to the XWIN System protocol.

1.3 Introduction to application development

This section is an introduction to the development of applications which can be run on a Reliant UNIX system and to the techniques for packaging and installing application software.

The topics covered include file and record locking, interprocess communication, symbolic links, virtual memory the process scheduler, data validation and modification of the *sysadm* interface.

1.3.1 Application development

Each application performs a different function, but goes through the same basic steps: input, processing, and output. This section briefly describes tools you can use to accomplish these steps. Then it refers you to other chapters in this book or to other documents for more details.

For the input and output steps, most applications interact with an end user at a terminal.

During the processing step, sometimes an application needs access to special services provided by the operating system (for example, to interact with the file system, control processes, manage memory, and more). Some of these services are provided through system calls and some through libraries of functions. (System calls are grouped by function in a later section of this book.) Some system call services and data validation tools are described in detail later in this book.

1.3.1.1 File and record locking

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

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

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

Another distinction that needs to be made is that between mandatory and advisory locking. Mandatory locking means that the discipline is enforced automatically for the system calls that read, write, or create files. This is done through a permission flag established by the file's owner (or the system administrator). Advisory locking means that the processes that use the file take the responsibility for setting and removing locks as needed. Thus, mandatory may sound like a simpler and better deal, but it isn't so. The mandatory locking capability is included in the system to comply with an agreement with *EUUG*, a European organization that represents the interests of UNIX system users. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term atomic is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such

a circumstance is to make certain the lock is in place before any I/O starts, and that it is not removed until the transaction is done. That calls for locking of the advisory variety.

Where to find more information

There is an example of file and record locking in the sample application in Section "liber, A Library System". The Chapter "Application programming in the Reliant UNIX system environment" in this book is a detailed discussion of the subject with a number of examples. The manual pages that apply to this facility are *fcntl(2)*, *fcntl(5)*, *lockf(3)*, and *chmod(2)* in the "Programmer's Reference Manual" [9]. *fcntl(2)* is the system call for file and record locking (although it isn't limited to that only). *fcntl(5)* tells you the file control options. The subroutine *lockf(3)* can also be used to lock sections of a file or an entire file. Setting *chmod* so that all portions of a file are locked will ensure that parts of files are not corrupted.

1.3.1.2 Interprocess communications

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

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

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. (Semaphores are typically used to maintain order among processes). These three terms define three different styles of communication between processes:

messages
: Communication is in the form of data stored in a buffer. The buffer can be either sent or received.

semaphores
: Communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25.

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

The sets of IPC system calls are:

msgget□□semget□□shmget□□msgctl□□semctl□□shmctl□□msgop□□semop□□shmop

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

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

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

Where to find more information

The Chapter "Application programming in the Reliant UNIX system environment" in this book gives a detailed description of IPC, with many code examples that use the IPC system calls. An example of the use of some IPC features is included in the *liber* application in Section "liber, A Library System". The system calls are described in Section 2 of [9].

1.3.1.3 Process scheduler

The Reliant UNIX system scheduler determines when processes run. It maintains process priorities based on configuration parameters, process behavior, and user requests; it uses these priorities to assign processes to the CPU.

By default, the scheduler uses a time-sharing policy. A time-sharing policy adjusts process priorities dynamically in an attempt to give good response time to interactive processes and good throughput to CPU-intensive processes.

The scheduler offers a real-time scheduling policy as well as a time-sharing policy.

Where to find more information

The Chapter "Application programming in the Reliant UNIX system environment", gives detailed information on the process scheduler, along with relevant code examples. See also *priocntl*(1) in [6], *priocntl*(2) in [9], and *dispadmin*(1M) in [7].

1.3.1.4 Symbolic links

A symbolic link is a special type of file that represents another file. The data in a symbolic link consists of the path name of a file or directory to which the symbolic link file refers. The link that is formed is called symbolic to distinguish it from a regular (hard) link. A symbolic link differs functionally from a regular link in three major ways.

1. A symbolic link can refer to a file from a different file system.
2. A symbolic link can refer to directories, as well as regular files.
3. A symbolic link can be created even if the file it represents does not exist.

When a user creates a regular link to a file, a new directory entry is created containing a new file name and the inode number of an existing file. The link count of the file is incremented.

In contrast, when a user creates a symbolic link, (using the *ln(1)* command with the -s option) both a new directory entry and a new inode are created. A data block is allocated to contain the path name of the file to which the symbolic link refers. The link count of the referenced file is not incremented.

Symbolic links can be used to solve a variety of common problems. For example, it frequently happens that a file system (such as /) runs out of disk space. With symbolic links, an administrator can create a link from a directory to a directory on another file system to make use of free disk space there. In most cases, this is transparent to both users and programs.

Symbolic links can also help maintain compatibility in spite of built-in path names. Changing such path names might necessitate changing the programs and recompiling them. With symbolic links, the path names can effectively be changed by making the original files symbolic links that point to new files.

In a shared resource environment, symbolic links can be very useful. For example, if it is important to have a single copy of certain administrative files, symbolic links can be used to help share them. Symbolic links can also be used to share resources selectively. Suppose a system administrator wants to do a remote mount of a directory that contains sharable devices. These devices must be in */dev* on the client system, but this system has devices of its own so the administrator does not want to mount the directory onto */dev*. Rather than do this, the administrator can mount the directory at a location other than */dev* and then use symbolic links in the */dev* directory to refer to these remote devices. (This is similar to the problem of built-in path names since it is normally assumed that devices reside in the */dev* directory.)

Where to find more information

The Chapter "Application programming in the Reliant UNIX system environment", discusses symbolic links in detail. Refer to *symlink*(2) in [9] for information on creating symbolic links. See also *stat*(2), *rename*(2), *link*(2), *readlink*(2), and *unlink*(2) in the same manual.

1.3.1.5 Memory management

The Reliant UNIX system includes a complete set of memory-mapping mechanisms. Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated.

The memory-management facilities

- unify the system's operations on memory
- provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services
- maintain consistency with the existing environment, in particular using the Reliant UNIX file system as the name space for named virtual-memory objects

The system's virtual memory consists of all available physical memory resources including local and remote file systems, processor primary memory, swap space, and other random-access devices. Named objects in the virtual memory are referenced though the Reliant UNIX file system. However, not all file system objects are in the virtual memory; devices that Reliant UNIX cannot treat as storage, such as terminal and network device files, are not in the virtual memory. Some virtual memory objects, such as private process memory and shared memory segments, do not have names.

The memory mapping interface

The applications programmer gains access to the facilities of the virtual memory system through several sets of system calls.

mmap
: establishes a mapping between a process's address space and a virtual memory object.

mprotect
: assigns access protection to a block of virtual memory

munmap
: removes a memory mapping

mincore
: tells whether mapped memory pages are in primary memory

Where to find more information

The Chapter "Application programming in the Reliant UNIX system environment", gives a detailed description of the virtual memory system. Refer to *mmap*(2), *mprotect*(2), *munmap*(2), and *mincore*(2) in [9] for these manual pages.

1.3.1.6 Data validation tools

Data validation tools are written to help you write any administrative programs and routines that are part of your software package. They help standardize the appearance of administration interaction in the Reliant UNIX system environment and also simplify development of scripts and programs requiring administrator input.

There are two types of data validation tools:

1. shell commands (to be used in shell scripts)
2. visual tools (to be used in FMLI form definitions)

The shell commands perform a series of tasks; the visual tools perform a subsection of the full series. These tasks are:

- prompting a user for input
- validating the answer
- printing a help message when requested
- presenting an error message when validation fails
- returning the input if it passes validation
- allowing a user to quit the process

Where to find more information

The Chapter "Application programming in the Reliant UNIX system environment", describes the characteristics of these tools and introduces you to the available tools. For details on a specific tool, refer to

the Chapter "Reference pages", which includes the manual pages for *ckdate*(1), *ckgid*(1), *ckint*(1), *ckkeywd*(1), *ckpath*(1), *ckrange*(1), *ckstr*(1), *cktime*(1), *ckuid*(1), *ckyorn*(1), *dispgid*(1), and *dispuid*(1).

1.3.2 Package development and installation

This section gives the software package developer information on the interfaces provided by Reliant UNIX, specifically package software for Reliant UNIX and how to modify the *sysadm* interface.

The *sysadm* interface modification tools allow you to generate files to deliver as part of your package. When these files are installed, your package administration tasks are added to the interface.

1.3.2.1 Packaging application software

Packaging software that will be installed on a computer running Reliant UNIX differs from packaging in a pre-Version 5.4x Reliant UNIX system environment. Pre-5.4x packages deliver information to the system through script actions, but packages for Reliant UNIX and above do this through package information files.

A software package is made up of a group of components that together create the software. These components naturally include the executables that comprise the software, but they also include at least two information files and can optionally include other information files and scripts.

The contents of a package fall into three categories:

1. required components
2. optional package information files
3. optional package scripts

A packaging tool, the *pkgmk* command, is provided to help automate package creation. It gathers the components of a package on the development machine and copies and formats them onto the installation medium.

The installation tool, the *pkgadd* command, copies the package from the installation medium onto the target system and there performs system housekeeping routines that concern the package.

Where to find more information

The Chapter "Packaging application and driver software", gives full details on packaging application software. In Section "Package installation case studies" examples are provided. For details on a specific tool, refer to Chapter "Reference pages", which includes the manual pages for *admin*(4), *compver*(4), *copyright*(4), *depend*(4), *installf*(1M), *pkgadd*(1M), *pkgask*(1M), *pkgchk*(1M), *pkginfo*(1), *pkginfo*(4), *pkginstallf*(1M), *pkgmap*(4), *pkgmk*(1), *pkgparam*(1), *pkgproto*(1), *pkgrm*(1M), *pkgtrans*(1), *prototype*(4), *removef*(1M), and *space*(4).

1.3.2.2 Modifying the sysadm interface

The Reliant UNIX system provides a menu interface to the most common administrative procedures. It is invoked by executing *sysadm* and is referred to as the *sysadm* interface.

You can deliver additions or changes to this interface as part of your application software package. Creating the necessary information for a *sysadm* interface modification can be done using the tools Reliant UNIX provides.

Two commands can be used to modify the interface. *edsysadm* allows you to make changes or additions to the interface. It is interactive (much like the *sysadm* command itself) and presents a series of prompts for information. Which prompts appear depend on your response to them. The *delsysadm* command deletes menus or tasks from the interface. In addition to these commands, a group of data validation tools are provided to simplify and standardize the programming of administrative interaction.

When you execute *edsysadm* to define menus and tasks and save those definitions to be included in your application software package, it creates the package description file, the menu information file, and a prototype file.

- The package description file contains information used by *edsysadm* to change interface modifications already saved for packaging.
- The menu information file contains the name of a menu and where it is located in the interface structure or the name of a task and what executable to use when the task is invoked.

- The prototype file created by *edsysadm* contains entries for all of the interface modification components that must be packaged with your software (for example, the menu information file and, for tasks, the executables).

You must take a number of steps if you intend to modify the *sysadm* interface by adding the administration to your package. You have to

- plan your package administration
- write your administration actions
- write your help messages
- package your interface modifications

Where to find more information

The Chapter "Packaging application and driver software", gives full details on modifying the *sysadm* interface. For details on a specific tool, refer to Chapter "Reference pages", which includes the manual pages for *delsysadm*(1M) and *edsysadm*(1M). The "System Administrator's Guide" [8] gives a complete description of the *sysadm* interface and how to use it. See also [15] for complete information on FMLI.

2 Application programming in the Reliant UNIX system environment

This part of the manual provides you with information on how to program applications in a Reliant UNIX system environment. In the following sections you will find

- an overview of system calls
- notes on process generation
- information on the process scheduler
- information on signals
- descriptions of the interprocess communication mechanisms

2.1 Overview of system calls

This section is an overview of system calls and other system services that you may need when developing and packaging application programs. The first subsection deals with error handling. This is followed by lists grouping together various types of system call and system service. For further information on system calls and system services refer to [9].

Reliant UNIX system calls are the interface between the kernel and the user programs that run on top of it. The system calls described in [9] define what the Reliant UNIX system is. Everything else is built on their foundation. Strictly speaking, they are the only way to access such facilities as the file system, interprocess communication primitives, and multitasking mechanisms.

Of course, most programs do not need to invoke system calls directly to gain access to these facilities. If you are writing a C program, for example, you can use the library functions described in [9] instead. When you use these functions, the details of their implementation on the Reliant UNIX system are transparent to the program, for example, that the system call *read* underlies the *fread* implementation in the standard C library. In other words, the program will generally be portable to any system, Reliant UNIX or not, with a conforming C implementation. (See Chapter 2 of [3] for a discussion of the standard C library.)

In contrast, programs that invoke system calls directly are portable only to other Reliant UNIX or Reliant UNIX-like systems; for that reason, you would not use *read* and *write* in a program that performed a simple input/output operation. Other operations, however, including most multitasking mechanisms, do require direct interaction with the Reliant UNIX system kernel.

A C program is automatically linked with the system calls you have invoked when you compile the program. The procedure may be different for programs written in other languages. Refer to [3] for details on the language you are using.

2.1.1 Error handling

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

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

The value in *errno* is not cleared on successful calls, so your program should check it only if the system call returned a −1. The errors are described in [9].

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

2.1.2 Basic file I/O

These system calls perform basic operations on Reliant UNIX system files.

Function name(s)	Purpose

open	open a file for reading and/or writing
close	close a file
read	read from a file
write	write to a file
creat	create a new file or rewrite an existing one
unlink	remove directory entry for file or decrement link counter
lseek	move read/write file pointer

Table 1: Basic file I/O system calls

2.1.3 Advanced file I/O

These system calls allow creation of new directories (and other things), linking to existing files, and obtaining or modifying file status information.

Function name(s)	Purpose
link	link to a file
access	determine file permissions
mknod	make a special file or FIFO
chmod, fchmod	change mode of file
chown, lchown, fchown	change owner and group of a file
utime	set file access and modification times
stat, lstat, fstat	get file status
fcntl	file control, file locking
ioctl	device control
fpathconf, pathconf	get configurable path name variables
getdents	read directory entries and put in file system-independent format
mkdir	make a directory
readlink	read the value of a symbolic link
rename	change the name of a file
rmdir	remove a directory
symlink	make a symbolic link to a file

Table 2: Advanced file I/O system calls

2.1.4 Terminal I/O

The following calls deal with a general terminal interface that is provided to control asynchronous communications ports.

Function name(s)	Purpose
tcgetattr, tcsetattr	get and set terminal attributes
tcsendbreak,	line control functions

tcdrain, tcflush, tcflow	
cfgetospeed, cfgetispeed, cfsetispeed, cfsetospeed	get and set baud rate functions
tcgetpgrp, tcsetpgrp	get and set terminal foreground process group ID
tcgetsid	get terminal session ID

Table 3: Terminal I/O system calls

2.1.5 Processes

These system calls control user processes.

Function name(s)	Purpose
fork	create a new process
exec, execl, execv, execle, execve, execlp, execvp	execute a file
exit, _exit	terminate process
wait	wait for child process to stop or terminate
setuid, setgid	set user and group IDs
setpgrp	set process group ID
chdir, fchdir	change working directory
chroot	change root directory
nice	change priority of a process
getcontext, setcontext	get and set current user context
getgroups, setgroups	get or set supplementary group access list IDs
getpid, getpgrp, getppid, getpgid	get process, process group, and parent process IDs
getuid, geteuid, getgid, getegid	get real and effective user and group IDs
pause	suspend process until signal
priocntl	process scheduler control
setpgid	set process group ID
setsid	set session ID
waitpid	wait for a child process to terminate
kill	send a signal to a process or group of processes

Table 4: Process system calls

2.1.6 Basic interprocess communication

These system calls connect processes so they can communicate. pipe is the system call for creating an interprocess channel. dup is the call for duplicating an open file descriptor. (These IPC mechanisms are

applicable only to processes which are associated with each other (*fork*).

2.1.7 Advanced interprocess communication

These system calls support interprocess messages, semaphores, and shared memory.

Function name(s)	Purpose
msgget	get message queue
msgctl	message control operations
msgop	message operations
semget	get set of semaphores
semctl	semaphore control operations
semop	semaphore operations
shmget	get shared memory segment identifier
shmctl	shared memory control operations
shmop	shared memory operations

Table 5: Advanced interprocess communication system calls

2.1.8 Memory management

These system calls allow you to control, monitor and manage virtual memory.

Function name(s)	Purpose
memcntl	memory management control
mmap	map pages of memory
mprotect	set protection of memory mapping
munmap	unmap pages of memory
plock	lock process in memory
brk, sbrk	change process address space

Table 6: Memory management system calls

2.1.9 File system control

These system calls allow you to control and monitor file systems.

Function name(s)	Purpose
sync	force updating of data on disk
mount,unmount	mount/unmount a file system
statfs, fstatfs	get file system information
sysfs	check file system type and index

Table 7: File system control system calls

2.1.10 Signals

Signals are messages passed to running processes.

Function name(s)	Purpose
sigaction	extended signal management
sigaltstack	set signal alternate stack context

signal, sigset, sighold, sigrelse, sigignore, sigpause	simplified signal management
sigpending	examine pending signals
sigprocmask	change or examine signal mask
sigsend, sigsendset	send a signal to a process or group of processes
sigsuspend	install a signal mask and suspend process until signal

Table 8: Signal system calls

2.1.11 Miscellaneous system calls

These are system calls for such things as administration, timing, and other miscellaneous purposes.

Function names(s)	Purpose
ulimit	get and set user limits
alarm	set a process alarm clock
getmsg	get next message off a stream
getrlimit, setrlimit	control maximum system resource consumption
uname	get/set name of current Reliant UNIX system
putmsg	send a message on a stream
profil	execution time profile
sysconf	determine value of configurable system variable
uadmin	reboot system
time	get time
stime	set time
acct	enable or disable process accounting

Table 9: Miscellaneous system calls

2.2 Process generation

Whenever you execute a command in the Reliant UNIX system you are initiating a process that is numbered (with a unique process identification number or *pid*) and tracked by the operating system. A flexible feature of the Reliant UNIX system is that processes can be generated by other processes. This happens more than you might ever be aware of. For example, when you log in to your system you are running a process, very probably the shell. If you then use an editor such as vi, take the option of invoking the shell from *vi*, and execute the *ps* command, you will see a display which shows the results of a *ps –f* command:

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

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

The four processes in the example above are all Reliant UNIX system shell-level commands, but you can spawn new processes from your own program. You might think, "Well, it's one thing to switch from one

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

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

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

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

exec(2)
: stop this process and start another

fork(2)
: start an additional copy of this process

2.2.1 exec(2)

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

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

For *execl* the argument list is

/usr/bin/prog2
: path name of the program to be executed in the new process

prog
: the name the new process gets in its *argv[0]*

progarg1,2
: arguments to *prog2* as *char* *'s

char *0
: a null *char* pointer to mark the end of the arguments

Refer to the appropriate entry in [9] for details on *exec*. The functions of the *exec* family supply no return value on successful execution: the new process overlays the process that makes the *exec* system call.

The new process also takes over the process ID and other attributes of the old process. If the call to *exec* is unsuccessful, control is returned to your program with a return value of –1. You can check *errno* to learn why it failed.

2.2.2 fork(2)

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

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

Your code might include statements like those in Figure "Example of fork". Keep in mind that the fragment of

code in this figure includes a minimum amount of checking for error conditions. There is also potential confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the *fork* or *exec* has the three standard files that are automatically opened: *stdin*, *stdout*, and *stderr*. If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the *fork*. Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process the pointer has moved on.

Because the child process ID is taken over by the new *exec*'d process, the parent knows the ID. What this boils down to is a way of leaving one program to run another, returning to the point in the first program where processing left off. By the way, this is exactly what the *system* function in the standard C library does.

Example of fork

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

2.3 Process scheduler

The Reliant UNIX system scheduler determines when processes run. It maintains process priorities based on configuration parameters, process behavior, and user requests; it uses these priorities to assign processes to the CPU.

By default, the Reliant UNIX 5.43 scheduler uses a time-sharing policy like the policy used in previous releases. A time-sharing policy adjusts process priorities dynamically in an attempt to provide good response time to interactive processes and good throughput to processes that use a lot of CPU time.

The Reliant UNIX 5.43 scheduler offers a real-time scheduling policy as well as a time-sharing policy. Real-time scheduling allows users to set fixed priorities on a per-process basis. The highest-priority real-time user process always gets the CPU as soon as it is runnable, even if system processes are runnable. An application can therefore specify the exact order in which processes run. An application may also be written so that its real-time processes have a guaranteed response time from the system.

For most Reliant UNIX environments, the default scheduler configuration works well and no real-time processes are needed: administrators should not change configuration parameters or scheduler properties for

their processes. However, when the requirements for an application include strict timing constraints, real-time processes sometimes provide the only way to satisfy those constraints.

Real-time processes used carelessly can have a dramatic negative effect on the performance of the entire system.

This chapter is addressed to programmers who need more control over order of process execution than they get using default scheduler parameters.

Because changes in scheduler administration can affect scheduler behavior, programmers may also need to know something about scheduler administration. For administrative information on the scheduler, see [8] . There are also a few entries with information on scheduler administration in [7]:

dispadmin(1M)
: tells how to change scheduler configuration in a running system.

ts_dptbl(4)
: contain the time-sharing and real-time parameter tables that

t_dptbl(4)
: are used to configure the scheduler.

The rest of this section is organized as follows:

- The Section "Overview of the process scheduler", tells what the scheduler does and how it does it. It also introduces scheduler classes.
- The Section "Commands and function calls", describes and gives examples of the *priocntl(1)* command and the *priocntl(2)* and *priocntlset(2)* system calls, the user interface to scheduler services. The *priocntl* functions allow you to retrieve scheduler configuration information and to get or set scheduler parameters for a process or a set of processes.
- In Section "Interaction with other functions", the interactions between the scheduler and related functions are described.
- The Section "Performance", discusses scheduler latencies that some applications must be aware of and mentions some considerations other than the scheduler that application designers must take into account to ensure that their requirements are met.

2.3.1 Overview of the process scheduler

The following figure shows how the Reliant UNIX 5.43 process scheduler works:

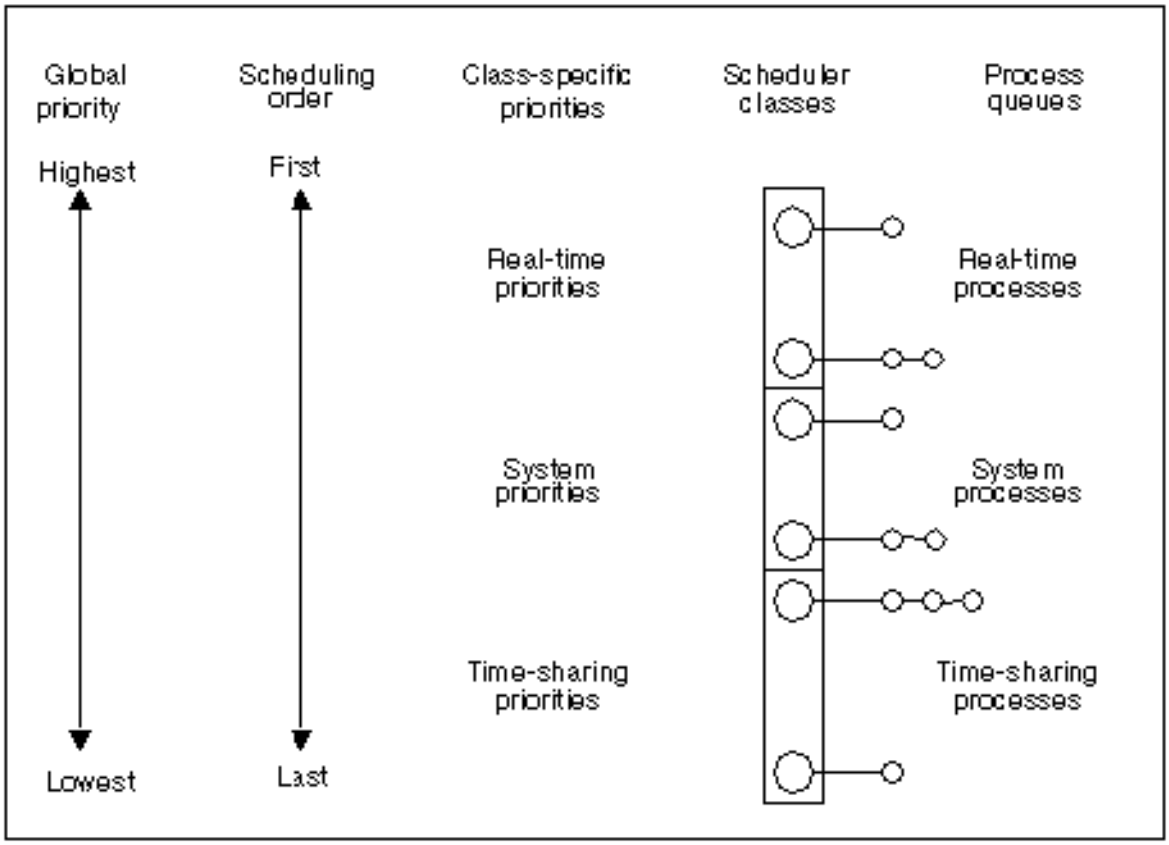


Figure 1: The Reliant UNIX 5.43 process scheduler

When a process is created, it inherits its scheduler parameters, including scheduler class and a priority within that class. A process changes class only as a result of a user request. The system manages the priority of a process based on user requests and a policy associated with the scheduler class of the process.

In the default configuration, the initialization process belongs to the time-sharing class. Because processes inherit their scheduler parameters, all user login shells begin as time-sharing processes in the default configuration.

The scheduler converts class-specific priorities into global priorities. The global priority of a process determines when it runs—the scheduler always runs the runnable process with highest global priority. Numerically higher priorities run first. Once the scheduler assigns a process to the CPU, the process runs until it uses up its time slice, or passes control away (waiting for I/O), or is preempted by a higher-priority process. Processes with the same priority run round-robin.

Administrators specify default time slices in the configuration tables, but users may assign per-process time slices to real-time processes.

You can display the global priority of a process with the *–cl* options of the *ps(1)* command. You can display configuration information about class-specific priorities with the *priocntl(1)* command and the *dispadmin(1M)* command.

By default, all real-time processes have higher priorities than any kernel process, and all kernel processes have higher priorities than any time-sharing process.

 As long as there is a runnable real-time process, no kernel process and no time-sharing process runs.

The following sections describe the scheduling policies of the three default classes.

2.3.1.1 Time-sharing class

The goal of the time-sharing policy is to provide good response time to interactive processes and good throughput to CPU-bound processes. The scheduler switches CPU allocation frequently enough to provide good response time. Time slices are typically on the order of a few hundred milliseconds.

The time-sharing policy changes priorities dynamically and assigns time slices of different lengths. The scheduler raises the priority of a process that sleeps after only a little CPU use (a process sleeps, for example, when it starts an I/O operation such as a terminal read or a disk read); frequent sleeps are characteristic of interactive tasks such as editing and running simple shell commands. On the other hand, the time-sharing policy lowers the priority of a process that uses the CPU for long periods without sleeping.

The default time-sharing policy gives larger time slices to processes with lower priorities. A process with a low priority is likely to be CPU-intensive.

Other processes get the CPU first, but when a low-priority process finally gets the CPU, it gets a bigger chunk of time. If a higher-priority process becomes runnable during a time slice, however, it preempts the running process.

The scheduler manages time-sharing processes using configurable parameters in the time-sharing parameter table *ts_dptbl*. This table contains information specific to the time-sharing class.

2.3.1.2 System class

The system class uses a fixed-priority policy to run kernel processes such as servers and housekeeping processes like the paging demon. The system class is reserved for use by the kernel; users may neither add nor remove a process from the system class. Priorities for system class processes are set up in the kernel code for those processes; once established, the priorities of system processes do not change. (User processes running in kernel mode are not in the system class.)

2.3.1.3 Real-time class

The real-time class uses a fixed-priority scheduling policy so that critical processes can run in predetermined order. Real-time priorities never change except when a user requests a change. Contrast this fixed-priority policy with the time-sharing policy, in which the system changes priorities in order to provide good interactive response time.

Privileged users can use the *priocntl* command or the *priocntl* system call to assign real-time priorities.

The scheduler manages real-time processes using configurable parameters in the real-time parameter table *rt_dptbl*. This table contains information specific to the real-time class.

2.3.2 Commands and function calls

Here is a programmer's view of default process priorities

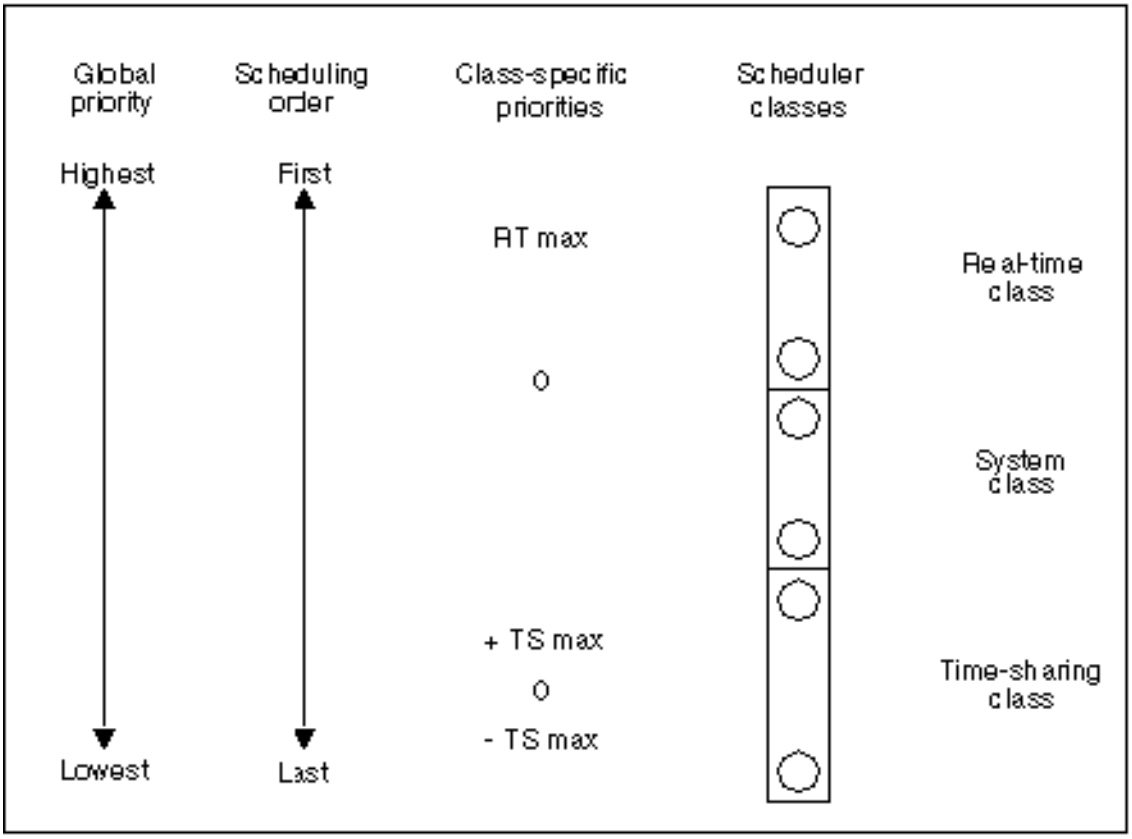


Figure 2: Process priorities (programmer view)

From a user or programmer's point of view, a process priority has meaning only in the context of a scheduler class. You specify a process priority by specifying a class and a class-specific priority value. The class and class-specific value are mapped by the system into a global priority that the system uses to schedule processes.

- Real-time priorities run from zero to a configuration-dependent maximum. The system maps them directly into global priorities. They never change except when a user changes them.
- System priorities are controlled entirely in the kernel. Users cannot affect them.
- Time-sharing priorities have a user-controlled component (the "user priority") and a component controlled by the system. The system does not change the user priority except as the result of a user request. The system changes the system-controlled component dynamically on a per-process basis in order to provide good overall system performance; users cannot affect the system-controlled component. The scheduler combines these two components to get the process global priority.

  The user priority runs from the negative of a configuration-dependent maximum to the positive of that maximum. A process inherits its user priority. Zero is the default initial user priority.

  The "user priority limit" is the configuration-dependent maximum value of the user priority. You may set a user priority to any value below the user priority limit. With appropriate permission, you may raise the user priority limit. Zero is the default user priority limit.

  You may lower the user priority of a process to give the process reduced access to the CPU or, with the appropriate permission, raise the user priority to get better service. Because you cannot set the user priority above the user priority limit, you must raise the user priority limit before you raise the user priority if both have their default values of zero.

  An administrator configures the maximum user priority independent of global time-sharing priorities. In the default configuration, for example, a user may set a user priority only in the range from –20 to +20, but 60 time-sharing global priorities are configured.

A system administrator's view of priorities is different from that of a user or programmer. When configuring scheduler classes, an administrator deals directly with global priorities. The system maps priorities supplied by users into these global priorities. Also refer to [8].

The *ps -cel* command reports global priorities for all active processes. The *priocntl* command reports the class-specific priorities that users and programmers use.

Global process priorities and user-supplied priorities are in ascending order: numerically higher priorities run first.

The *priocntl(1)* command and the *priocntl(2)* and *priocntlset(2)* system calls set or retrieve scheduler parameters for processes. The basic idea for setting priorities is the same for all three functions:

- Specify the target processes.
- Specify the scheduler parameters you want for those processes.
- Do the command or system call to set the parameters for the processes.

You specify the target processes using an ID type and an ID. The ID type tells how to interpret the ID. (This concept of a set of processes applies to signals as well as to the scheduler; see *sigsend(2)*.) The following table lists the valid ID types that you may specify.

priocntl ID types
process ID
parent process ID
process group ID
session ID
class ID
effective user ID
effective group ID
all processes

Table 10: Valid ID types

These IDs are basic properties of Reliant UNIX processes (see *intro(2)*). The class ID refers to the scheduler class of the process. *priocntl* works only for the time-sharing and the real-time classes, not for the system class. Processes in the system class have fixed priorities assigned when they are started by the kernel.

2.3.2.1 The priocntl command

The *priocntl* command comes in four forms:

priocntl -l
: displays configuration information.

priocntl -d
: displays the scheduler parameters of processes.

priocntl -s
: sets the scheduler parameters of processes.

priocntl -e
: executes a command with the specified scheduler parameters.

1. Here is the output of the *-l* option for the default configuration.

```
$ priocntl -l
CONFIGURED CLASSES
==================
SYS (System Class)
```

TS□(Time□Sharing)
□□□□□□□□□□□Configured□TS□User□Priority□Range:□-20□through□20
RT□(Real□Time)
□□□□□□□□□□□Maximum□Configured□RT□Priority:□59

2. The *–d* option displays the scheduler parameters of a process or a set of processes. The syntax for this option is

   priocntl -d -i idtype idlist

   idtype tells what kind of IDs are in *idlist*. *idlist* is a list of IDs separated by white space. Here are the valid values for *idtype* and their corresponding ID types in *idlist*:

idtype	idlist
pid	process IDs
ppid	parent process IDs
pgid	process group IDs
sid	session IDs
class	class names (TS or RT)
uid	effective user IDs
gid	effective group IDs
all	

Table 11: Valid values for ID types

   Here are some examples of the *–d* option of *priocntl*:

$□#□display□info□on□all□processes
$□priocntl□-d□-i□all
□□□□□□□□□□.
□□□□□□□□□□.
□□□□□□□□□□.
$□#□display□info□on□all□time-sharing□processes
$□priocntl□-d□-i□class□TS
□□□□□□□□□□.
□□□□□□□□□□.
□□□□□□□□□□.
$□#□display□info□on□all□processes□with□user□ID□103□or□6626
$□priocntl□-d□-i□uid□103□6626
□□□□□□□□□□.
□□□□□□□□□□.
□□□□□□□□□□.

3. The *–s* option sets scheduler parameters for a process or a set of processes. The syntax for this option is

   priocntl□-s□-c□class□□class_options□-i□idtype□□idlist

   idtype and *idlist* are the same as for the *–d* option described above.

   class is *TS* for time-sharing or *RT* for real-time. You must be system administrator to create a real-time process, to raise a time-sharing user priority above a per-process limit, or to raise the per-process limit above zero. Class options are class-specific:

Class	-c class	Options	Meaning
real-time	RT	-p pri-t tslc-r res	prioritytime sliceresolution
time-sharing	TS	-p upri-m	user prioriyuser priority

		uprilim	limit

Table 12: Class-specific options for priocntl

For a real-time process you may assign a priority and a time slice.

- The priority is a number from 0 to the real-time maximum as reported by *priocntl -l*; the default maximum is 59.
- You specify the time slice as a number of clock intervals and the resolution of the interval. Resolution is specified in intervals per second. The time slice, therefore, is *tslc/res* seconds. To specify a time slice of one-tenth of a second, for example, you could specify a *tslc* of 1 and a *res* of 10. If you specify a time slice without specifying a resolution, millisecond resolution (a *res* of 1000) is assumed.

If you change a time-sharing process into a real-time process, it gets a default priority and time slice if you don't specify one. If you wish to change only the priority of a real-time process and leave its time slice unchanged, omit the -t option. If you wish to change only the time slice of a real-time process and leave its priority unchanged, omit the *–p* option.

For a time-sharing process you may assign a user priority and a user priority limit.

- The user priority is the user-controlled component of a time-sharing priority. The scheduler calculates the global priority of a time-sharing process by combining this user priority with a system-controlled component that depends on process behavior. The user priority has the same effect as a value set by *nice* (except that *nice* uses higher numbers for lower priority).
- The user priority limit is the maximum user priority a process may set for itself without being system administrator. By default, the user priority limit is 0; you must be system administrator to set a user priority limit above 0.

Both the user priority and the user priority limit must be within the user priority range reported by the *priocntl -l* command. The default range is –20 to +20.

A process may lower and raise its user priority as often as it wishes, as long as the value is below its user priority limit. It is a courtesy to other users to lower your user priority for big chunks of low-priority work. On the other hand, if you lower your user priority limit, you must be system administrator to raise it. A typical use of the user priority limit is to reduce permanently the priority of child processes or of some other set of low-priority processes.

The user priority can never be greater than the user priority limit. If you set the user priority limit below the user priority, the user priority is lowered to the new user priority limit. If you attempt to set the user priority above the user priority limit, the user priority is set to the user priority limit.

Here are some examples of the *–s* option of *priocntl*:

```
#□#□make□process□with□ID□24668□a□real-time□process□with□default
#□#□parameters:
#□priocntl□-s□-c□RT□-i□pid□24668
#□#□make□3608□RT□with□priority□55□and□a□one-fifth□second□time□slice:
#□priocntl□-s□-c□RT□-p□55□-t□1□-r□5□-i□pid□3608
#□#□change□all□processes□into□time-sharing-processes:
#□priocntl□-s□-c□TS□-i□all
#□#□for□uid□1122,□reduce□TS□user□priority□and□user□priority□limit\□□□□□□□□□□to□-10:
#□priocntl□-s□-c□TS□-p□-10□-m□-10□-i□uid□1122
```

4. The *–e* option sets scheduler parameters for a specified command and executes the command. The syntax for this option is

   priocntl□-e□-c□class□□class_options□□command□□[command arguments]

   The class and class options are the same as for the *–s* option described above.

```
#□#□start□a□real-time□shell□with□default□real-time□priority:
#□priocntl□-e□-c□RT□/bin/sh
$□#□run□make□with□a□time-sharing□□user□priority□of□-10:
$□priocntl□-e□-c□TS□-p□-10□make□bigprog
```

The *priocntl* command subsumes the function of *nice* ,which continues to work as on previous releases. *nice* works only on time-sharing processes and uses higher numbers to assign lower priorities. The final example above is equivalent to using *nice* to set an "increment" of 10:

nice -10 make bigprog

2.3.2.2 The priocntl system call

```
#include<sys/types.h>
#include<sys/procset.h>
#include<sys/priocntl.h>
#include<sys/rtpriocntl.h>
#include<sys/tspriocntl.h>
long priocntl(idtype_t idtype, id_t id, int cmd,
      cmd_struct arg);
```

The *priocntl* system call gets or sets scheduler parameters of a set of processes. The input arguments:

- *idtype* is the type of ID you are specifying.
- *id* is the ID.
- *cmd* specifies which *priocntl* function to perform. The functions are listed in the table below.
- *arg* is a pointer to a structure that depends on *cmd*.

Here are the valid values for *idtype*, which are defined in *priocntl.h*, and their corresponding ID types:

idtype	Interpretation of id
P_PID	process ID (of a single process)
P_PPID	parent process ID
P_PGID	process group ID
P_SID	session ID
P_CID	class ID
P_UID	effective user ID
P_GID	effective group ID
P_ALL	all processes

Table 13: Definition of ID types in priocntl.h

Here are the valid values for *cmd*, their meanings, and the type of *arg*.

cmd	arg type	Function
PC_GETCID	pcinfo_t	get class ID and attributes
PC_GETCLINFO	pcinfo_t	get class name and attributes
PC_SETPARMS	pcparms_t	set class and scheduling parameters
PC_GETPARMS	pcparms_t	get class and scheduling parameter

Table 14: priocntl commands

Here are the values *priocntl* returns on success:

- The *GETCID* and *GETCLINFO* commands return the number of configured scheduler classes.
- *PC_SETPARMS* returns 0.
- *PC_GETPARMS* returns the process ID of the process whose scheduler properties it is returning.

On failure, *priocntl* returns –1 and sets *errno* to indicate the reason for the failure. See *priocntl(2)* for the complete list of error conditions.

PC_GETCID, PC_GETCLINFO

The *PC_GETCID* and *PC_GETCLINFO* commands retrieve scheduler parameters for a class based on the class ID or class name. Both commands use the *pcinfo* structure to send arguments and receive return values:

```
typedef struct pcinfo {
    id_t    pc_cid;                    /* class id */
    char    pc_clname[PC_CLNMSZ];      /* class name */
    long    pc_clinfo[PC_CLINFOSZ];    /* class information */
} pcinfo_t;
```

The *PC_GETCID* command gets scheduler class ID and parameters given the class name. The class ID is used in some of the other *priocntl* commands to specify a scheduler class. The valid class names are *TS* for time-sharing and *RT* for real-time.

For the real-time class, *pc_clinfo* contains an *rtinfo* structure, which holds *rt_maxpri*, the maximum valid real-time priority; in the default configuration, this is the highest priority any process can have. The minimum valid real-time priority is zero. *rt_maxpri* is a configurable value. [8] tells how to configure process priorities.

```
typedef struct rtinfo {
    short   rt_maxpri;  /* maximum real-time priority */
} rtinfo_t;
```

For the time-sharing class, *pc_clinfo* contains a *tsinfo* structure, which holds *ts_maxupri*, the maximum time-sharing user priority. The minimum time-sharing user priority is *--ts_maxupri*. *ts_maxupri* is also a configurable value.

```
typedef struct tsinfo {
    short   ts_maxupri;  /* limits of user priority range */
} tsinfo_t;
```

The following program is a cheap substitute for *priocntl –l*; it gets and prints the range of valid priorities for the time-sharing and real-time scheduler classes.

```
/*
 * Get scheduler class IDs and priority ranges.  */
 */
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
main ()
{
          pcinfo_t          pcinfo;
          tsinfo_t          *tsinfop;
          rtinfo_t          *rtinfop;
          short             maxtsupri, maxrtpri;
 /* time-sharing */
       (void) strcpy (pcinfo.pc_clname, "TS");
       if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
                perror ("PC_GETCID failed for time-sharing class");
                exit (1);
       }
       tsinfop = (struct tsinfo *) pcinfo.pc_clinfo;
       maxtsupri = tsinfop->ts_maxupri;
       (void) printf ("Time sharing: ID %ld, priority range -%d through %d\n",
```

```
□□□□□□□□□□□□□□□□pcinfo.pc_cid,□maxtsupri,□maxtsupri);
□/*□real-time□*/
□□□□□□(void)□strcpy(pcinfo.pc_clname,□"RT");
□□□□□□if□(priocntl□(0L,□0L,□PC_GETCID,□&pcinfo)□==□-1L)□{
□□□□□□□□□□□□□□□□perror□("PC_GETCID□failed□for□real-time□class");
□□□□□□□□□□□□□□□□exit□(2);
□□□□□□}
□□□□□□rtinfop□=□(struct□rtinfo□*)□pcinfo.pc_clinfo;
□□□□□□maxrtpri□=□rtinfop->rt_maxpri;
□□□□□□(void)□printf("Real□time:□□□□ID□%ld,□priority□range□0□through□%d\n",
□□□□□□□□□□□□□□□□pcinfo.pc_cid,□maxrtpri);
□□□□□□return□(0);
}
```

The following screen shows the output of this program, called *getcid* in this example.

```
$□getcid
Time□sharing:□ID□1,□priority□range□-20□through□20
Real□time:□□□□ID□2,□priority□range□0□through□59
```

The following function is useful in the examples below. Given a class name, it uses *PC_GETCID* to return the class ID and maximum priority in the class. All the following examples omit the lines that include header files. The examples compile with the same header files as in the first example above.

```
/*
*□□Return□class□ID□and□maximum□priority.
*□□Input□argument□name□is□class□name.
*□□Maximum□priority□is□returned□in□*maxpri.
*/
id_t
schedinfo□(name,□maxpri)
□□□□□□□□□□char□*name;
□□□□□□□□□□short□*maxpri;
{
□□□□□□□□□□pcinfo_t□□□□□□□□info;
□□□□□□□□□□tsinfo_t□□□□□□□□*tsinfop;
□□□□□□□□□□rtinfo_t□□□□□□□□*rtinfop;
□□□□□□□□□□(void)□strcpy(info.pc_clname,□name);
□□□□□□□□□□if□(priocntl□(0L,□0L,□PC_GETCID,□&info)□==□-1L)□{
□□□□□□□□□□□□□□□□□□□□return□(-1);
□□□□□□□□□□}
□□□□□□□□□□if□(strcmp(name,□"TS")□==□0)□{
□□□□□□□□□□□□□□□□□□□□tsinfop□=□(struct□tsinfo□*)□info.pc_clinfo;
□□□□□□□□□□□□□□□□□□□□*maxpri□=□tsinfop->ts_maxupri;
□□□□□□□□□□}□else□if□(strcmp(name,□"RT")□==□0)□{
□□□□□□□□□□□□□□□□□□□□rtinfop□=□(struct□rtinfo□*)□info.pc_clinfo;
□□□□□□□□□□□□□□□□□□□□*maxpri□=□rtinfop->rt_maxpri;
□□□□□□□□□□}□else□{
□□□□□□□□□□□□□□□□□□□□return□(-1);
□□□□□□□□□□}
□□□□□□□□□□□□□□□□□□□□return□(info.pc_cid);
}
```

The *PC_GETCLINFO* command gets a scheduler class name and parameters given the class ID. This command makes it easy to write applications that make no assumptions about what classes are configured.

The following program uses *PC_GETCLINFO* to get the class name of a process based on the process ID. This program assumes the existence of a function *getclassID*, which retrieves the class ID of a process given

the process ID; this function is given in the following section.

```
□/*□□Get□scheduler□class□name□given□process□ID.□*/
main□(argc,□argv)
□□□□□□□□□□int□argc;
□□□□□□□□□□char□*argv[];
{
□□□□□□□□□□pcinfo_t□□□□□□□□□pcinfo;
□□□□□□□□□□id_t□□□□□□□□□□□□□□pid,□classID;
□□□□□□□□□□id_t□□□□□□□□□□□□□□getclassID();
□□□□□□□□□□if□((pid□=□atoi(argv[1]))□<=□0)□{
□□□□□□□□□□□□□□□□□□□□□□perror□("bad□pid");
□□□□□□□□□□□□□□□□□□□□□□exit□(1);
□□□□□□□□□□}
□□□□□□□□□□if□((classID□=□getclassID(pid))□==□-1)□{
□□□□□□□□□□□□□□□□□□□□□□perror□("unknown□class□ID");
□□□□□□□□□□□□□□□□□□□□□□exit□(2);
□□□□□□□□□□}
□□□□□□□□□□pcinfo.pc_cid□=□classID;
□□□□□□□□□□if□(priocntl□(0L,□0L,□PC_GETCLINFO,□&pcinfo)□==□-1L)□{
□□□□□□□□□□□□□□□□□□□□□□perror□("PC_GETCLINFO□failed");
□□□□□□□□□□□□□□□□□□□□□□exit□(3);
□□□□□□□□□□}
□□□□□□□□□□(void)□printf("process□ID□%d,□class□%s\n",□pid,□pcinfo.pc clname);
}
```

PC_GETPARMS, PC_SETPARMS

The *PC_GETPARMS* command gets and the *PC_SETPARMS* command sets scheduler parameters for processes. Both commands use the *pcparms* structure to send arguments or receive return values:

```
typedef□struct□pcparms□{
□□□id_t□□pc_cid;□□□□□□□□□□□□□□□□□□□□/*□process□class□*/
□□□long□□pc_clparms[PC_CLPARMSZ];□□/*□class□specific□*/
}□pcparms_t;
```

Ignoring class-specific information for the moment, we can write a simple function for returning the scheduler class ID of a process, as promised in the previous section.

```
/*
□*□□Return□scheduler□class□ID□of□process□with□ID□pid.□□*/
*/
getclassID□(pid)
□□□□□□□□□□id_t□pid;
{
□□□□□□□□□□pcparms_t□□□□□□□□pcparms;
□□□□□□□□□□pcparms.pc_cid□=□PC_CLNULL;
□□□□□□□□□□if□(priocntl(P_PID,□pid,□PC_GETPARMS,□&pcparms)□==□-1)□{
□□□□□□□□□□□□□□□□□□□□□□return□(-1);
□□□□□□□□□□}
□□□□□□□□□□return□(pcparms.pc_cid);
}
```

For the real-time class, *pc_clparms* contains an *rtparms* structure. *rtparms* holds scheduler parameters specific to the real-time class:

```
typedef□struct□rtparms□{
□□□short□□rt_pri;□□□□□□□/*□real-time□priority□*/
□□□ulong□□rt_tqsecs;□□□□/*□seconds□in□time□quantum□*/
```

```
    long  rt_tqnsecs;  /* additional nsecs in quantum */
} rtparms_t;
```

rt_pri is the real-time priority; *rt_tqsecs* is the number of seconds and *rt_tqnsecs* is the number of additional nanoseconds in a time slice. That is, *rt_tqsecs* seconds plus *rt_tqnsecs* nanoseconds is the interval a process may use the CPU without sleeping before the scheduler gives another process a chance at the CPU.

For the time-sharing class, *pc_clparms* contains a *tsparms* structure. *tsparms* holds the scheduler parameter specific to the time-sharing class:

```
typedef struct tsparms {
    short  ts_uprilim;  /* user priority limit */
    short  ts_upri;     /* user priority */
} tsparms_t;
```

ts_upri is the user priority, the user-controlled component of a time-sharing priority. *ts_uprilim* is the user priority limit, the maximum user priority a process may set for itself without being the system administrator. These values are described above in the discussion of the *-s* option of the *priocntl* command. Both the user priority and the user priority limit must be within the range reported by the *priocntl -l* command; this range is also reported by the *PC_GETCID* and *PC_GETCLINFO* commands to the *priocntl* system call.

The *PC_GETPARMS* command gets the scheduler class and parameters of a single process. The return value of the *priocntl* is the process ID of the process whose parameters are returned in the *pcparms* structure. The process chosen depends on the *idtype* and *id* arguments to *priocntl* and on the value of *pcparms.pc_cid*, which contains *PC_CLNULL* or a class ID returned by *PC_GETCID*:

Number of processes selected by idtype and id	**pc_c id**		
	RT class ID	**TS class ID**	**PC_CLNULL**
1	RT parameters of process selected	TS parameters of process selected	class and parameters of process selected
more than 1	RT parameters of highest priority RT process	TS parameters of process with highest user priority	(error)

Table 15: What gets returned by PC_GETPARMS

If *idtype* and *id* select a single process and *pc_cid* does not conflict with the class of that process, *priocntl* returns the scheduler parameters of the process. If they select more than one process of a single scheduler class, *priocntl* returns parameters using class-specific criteria as shown in the table. *priocntl* returns an error in the following cases:

- *idtype* and *id* select one or more processes and none is in the class specified by *pc_cid*.
- *idtype* and *id* select more than one process process and *pc_cid* is *PC_CLNULL*.
- *idtype* and *id* select no processes.

The following program takes a process ID as its input and prints the scheduler class and class-specific parameters of that process:

```
/*
 * Get scheduler class and parameters of
 * process whose pid is input argument.  *
 */
main (argc, argv)
          int argc;
```

```
          char *argv[];
{
          pcparms_t       pcparms;
          rtparms_t       *rtparmsp;
          tsparms_t       *tsparmsp;
          id_t            pid, rtID, tsID;
          id_t            schedinfo();
          short           priority, tsmaxpri, rtmaxpri;
          ulong           secs;
          long            nsecs;
          pcparms.pc_cid = PC_CLNULL;
          rtparmsp = (rtparms_t *) pcparms.pc_clparms;
          tsparmsp = (tsparms_t *) pcparms.pc_clparms;
          if ((pid = atoi(argv[1])) <= 0) {
                    perror ("bad pid");
                    exit (1);
          }
   /* get scheduler properties for this pid*/
          if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
                    perror ("GETPARMS failed");
                    exit (2);
          }
   /* get class IDs and maximum priorities for TS and RT */
          if ((tsID = schedinfo ("TS", &tsmaxpri)) == -1) {
                    perror ("schedinfo failed for TS");
                    exit (3);
          }
          if ((rtID = schedinfo ("RT", &rtmaxpri)) == -1) {
                    perror ("schedinfo failed for RT");
                    exit (4);
          }
   /* print results*/
          if (pcparms.pc_cid == rtID) {
                    priority = rtparmsp->rt_pri;
                    secs = rtparmsp->rt_tqsecs;
                    nsecs =  rtparmsp->rt_tqnsecs;
                    (void) printf ("process %d: RT priority %d\n",
                              pid, priority);
                    (void) printf ("      time slice %ld secs, %ld nsecs\n",
                              secs, nsecs);
          } else if (pcparms.pc_cid == tsID) {
                    priority = tsparmsp->ts_upri;
                    (void) printf ("process %d: TS priority %d\n",
                              pid, priority);
          } else {
                    printf ("Unknown scheduler class %d\n",
                              pcparms.pc_cid);
                    exit (5);
          }
          return (0);
}
```

The *PC_SETPARMS* command sets the scheduler class and parameters of a set of processes. The *idtype* and *id* input arguments specify the processes to be changed. The *pcparms* structure contains the new parameters:

pc_cid contains the ID of the scheduler class to which the processes are to be assigned, as returned by *PC_GETCID*; *pc_clparms* contains the class-specific parameters:

- If *pc_cid* is the real-time class ID, *pc_clparms* contains an *rtparms* structure in which *rt_pri* contains the real-time priority and *rt_tqsecs* plus *rt_tqnsecs* contains the time slice to be assigned to the processes.
- If *pc_cid* is the time-sharing class ID, *pc_clparms* contains a *tsparms* structure in which *ts_uprilim* contains the user priority limit and *ts_upri* contains the user priority to be assigned to the processes.

The following program takes a process ID as input, makes the process a real-time process with the highest valid priority minus 1, and gives it the default time slice for that priority. The program calls the *schedinfo* function listed above to get the real-time class ID and maximum priority.

```
/*
 * Input arg is proc ID.  Make process a real-time
 * process with highest priority minus 1.  *
 */
main (argc, argv)
          int argc;
          char *argv[];
{
          pcparms_t       pcparms;
          rtparms_t       *rtparmsp;
          id_t            pid, rtID;
          id_t            schedinfo();
          short           maxrtpri;
          if ((pid = atoi(argv[1])) <= 0) {
                    perror ("bad pid");
                    exit (1);
          }
   /* get highest valid RT priority */
          if ((rtID = schedinfo ("RT", &maxrtpri)) == -1) {
                    perror ("schedinfo failed for RT");
                    exit (2);
          }
   /*  change proc to RT, highest prio -1, default time slice */
          pcparms.pc_cid = rtID;
          rtparmsp = (struct rtparms *) pcparms.pc_clparms;
          rtparmsp->rt_pri = maxrtpri -1;
          rtparmsp->rt_tqnsecs = RT_TQDEF;
          if (priocntl(P_PID, pid, PC_SETPARMS, &pcparms) == -1) {
                    perror ("PC_SETPARMS failed");
                    exit (3);
          }
}
```

The following table lists the special values *rt_tqnsecs* can take when *PC_SETPARMS* is used on real-time processes. When any of these is used, *rt_tqsecs* is ignored. These values are defined in the header file *rtpriocntl.h*:

rt_tqnsecs	Time slice
RT_TQINF	infinite
RT_TQDEF	default
RT_NOCHANGE	unchanged

Table 16: rt_tqnsecs values for using PC_SETPARMS

RT_TQINF specifies an infinite time slice. *RT_TQDEF* specifies the default time slice configured for the

real-time priority being set with the *SETPARMS* call. *RT_NOCHANGE* specifies no change from the current time slice; this value is useful, for example, when you change process priority but do not wish to change the time slice. (You can also use *RT_NOCHANGE* in the *rt_pri* field to change a time slice without changing the priority.)

2.3.2.3 The priocntlset system call

```
#include□□□□□□□□<sys/types.h>
#include□□□□□□□□<sys/signal.h>
#include□□□□□□□□<sys/procset.h>
#include□□□□□□□□<sys/priocntl.h>
#include□□□□□□□□<sys/rtpriocntl.h>
#include□□□□□□□□<sys/tspriocntl.h>
long□priocntlset(procset_t□*psp,□int□cmd,
□□□□□□cmd_struct□arg);
```

The *priocntlset* system call changes scheduler parameters of a set of processes, just like *priocntl*. *priocntlset* has the same command set as *priocntl*; the *cmd* and *arg* input arguments are the same. But while *priocntl* applies to a set of processes specified by a single *idtype/id* pair, *priocntlset* applies to a set of processes that results from a logical combination of two *idtype/id* pairs.

The input argument *psp* points to a *procset* structure that specifies the two *idtype/id* pairs and the logical operation to perform. This structure is defined in *procset.h*:

```
typedef□struct□procset□{
□□□□□□□□idop_t□□□□p_op;□□□□□□□□□□/*□operator□connecting□*/
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□/*□left□and□right□sets□*/
/*□left□set:□□*/
□□□□□□□□idtype_t□□p_lidtype;□□□□/*□left□ID□type□*/
□□□□□□□□id_t□□□□□□p_lid;□□□□□□□□/*□left□ID□*/
□/*□right□set:□□*/
□□□□□□□□idtype_t□□p_ridtype;□□□□/*□right□ID□type□*/
□□□□□□□□id_t□□□□□□p_rid;□□□□□□□□/*□right□ID□*/
□}□procset_t;
```

p_lidtype and *p_lid* specify the ID type and ID of one ("left") set of processes; *p_ridtype* and *p_rid* specify the ID type and ID of a second ("right") set of processes. *p_op* specifies the operation to perform on the two sets of processes to get the set of processes to operate on. The valid values for *p_op* and the processes they specify are:

POP_DIFF
: set difference - processes in left set and not in right set

POP_AND
: set intersection - processes in both left and right sets

POP_OR
: set union - processes in either left or right sets or both

POP_XOR
: set exclusive-or - processes in left or right set but not in both

The following macro, also defined in *procset.h*, offers a convenient way to initialize a *procset* structure:

```
#define□setprocset(psp,□op,□ltype,□lid,□rtype,□rid)□\
□□□□□□□□□□□□□□□□(psp)->p_op□□□□□□□□=□(op);□\
□□□□□□□□□□□□□□□□(psp)->p_lidtype□□=□(ltype);□\
□□□□□□□□□□□□□□□□(psp)->p_lid□□□□□□=□(lid);□\
□□□□□□□□□□□□□□□□(psp)->p_ridtype□□=□(rtype);□\
□□□□□□□□□□□□□□□□(psp)->p_rid□□□□□□=□(rid);
```

Here is a situation where *priocntlset* would be useful: suppose an application had both real-time and time-sharing processes that ran under a single user ID. If the application wanted to change the priority of only

its real-time processes without changing the time-sharing processes to real-time processes, it could do so as follows. (This example uses the function *schedinfo*, which is defined above in the section on *PC_GETCID*.)

```
/*
 *  Change real-time priorities of this uid
 *  to highest real-time priority minus 1.
 */
main (argc, argv)
          int argc;
          char *argv[];
{
          procset_t        procset;
          pcparms_t        pcparms;
          struct rtparms  *rtparmsp;
          id_t             rtclassID;
          id_t             schedinfo();
          short            maxrtpri;
   /* left set: select processes with same uid as this process */
          procset.p_lidtype = P_UID;
          procset.p_lid = getuid();
   /* get info on real-time class */
          if ((rtclassID = schedinfo ("RT", &maxrtpri)) == -1) {
                    perror ("schedinfo failed");
                    exit (1);
          }
   /* right set: select real-time processes */
          procset.p_ridtype = P_CID;
          procset.p_rid = rtclassID;
   /* select only my RT processes */
          procset.p_op = POP_AND;
/*
   /* specify new scheduler parameters */
          pcparms.pc_cid = rtclassID;
          rtparmsp = (struct rtparms *) pcparms.pc_clparms;
          rtparmsp->rt_pri = maxrtpri - 1;
          rtparmsp->rt_tqnsecs = RT_NOCHANGE;
          if (priocntlset (&procset, PC_SETPARMS, &pcparms) == -1) {
                    perror ("priocntlset failed");
                    exit (2);
          }
}
```

priocntl offers a simple scheduler interface that is adequate for many applications; applications that need a more powerful way to specify sets of processes can use *priocntlset*.

2.3.3 Interaction with other functions

2.3.3.1 Kernel processes

The kernel assigns its demon and housekeeping processes to the system scheduler class. Users may neither add processes to nor remove processes from this class, nor may they change the priorities of these processes. The command *ps –cel* lists the scheduler class of all processes. Processes in the system class are identified by a *SYS* entry in the *CLS* column.

If the workload on a machine contains real-time processes that use too much CPU, they can lock out system processes, which can lead to all sorts of trouble. Real-time applications must ensure that they leave some CPU time for system and other processes.

2.3.3.2 fork, exec

Scheduler class, priority, and other scheduler parameters are inherited across the *fork(2)* and *exec(2)* system calls.

2.3.3.3 nice

The *nice(1)* command and the *nice(2)* system call work as in previous versions of the Reliant UNIX system. They allow you to change the priority of only a time-sharing process. You still use use lower numeric values to assign higher time-sharing priorities with these functions. To change the scheduler class of a process or to specify a real-time priority, you must use one of the *priocntl* functions. You use higher numeric values to assign higher priorities with the *priocntl* functions.

2.3.3.4 init

The *init* process is treated as a special case by the scheduler. To change the scheduler properties of *init*, *init* must be the only process specified by *idtype* and *id* or by the *procset* structure.

2.3.4 Performance

Because the scheduler determines when and for how long processes run, it has an overriding importance in the performance and perceived performance of a system.

By default, all processes are time-sharing processes. A process changes class only as a result of one of the *priocntl* functions.

In the default configuration, all real-time process priorities are above any time-sharing process priority. This implies that as long as any real-time process is runnable, no time-sharing process or system process ever runs. So if a real-time application is not written carefully, it can completely lock out users and essential kernel housekeeping.

Besides controlling process class and priorities, a real-time application must also control several other factors that influence its performance. The most important factors in performance are CPU power, amount of primary memory, and I/O throughput. These factors interact in complex ways. For more information, see the chapter on performance management in [8]. In particular, the *sar(1)* command has options for reporting on all the factors discussed in this section.

2.3.4.1 Process state transition

Applications that have strict real-time constraints may need to prevent processes from being swapped or paged out to secondary memory. Here's a simplified overview of Reliant UNIX process states and the transitions between states:

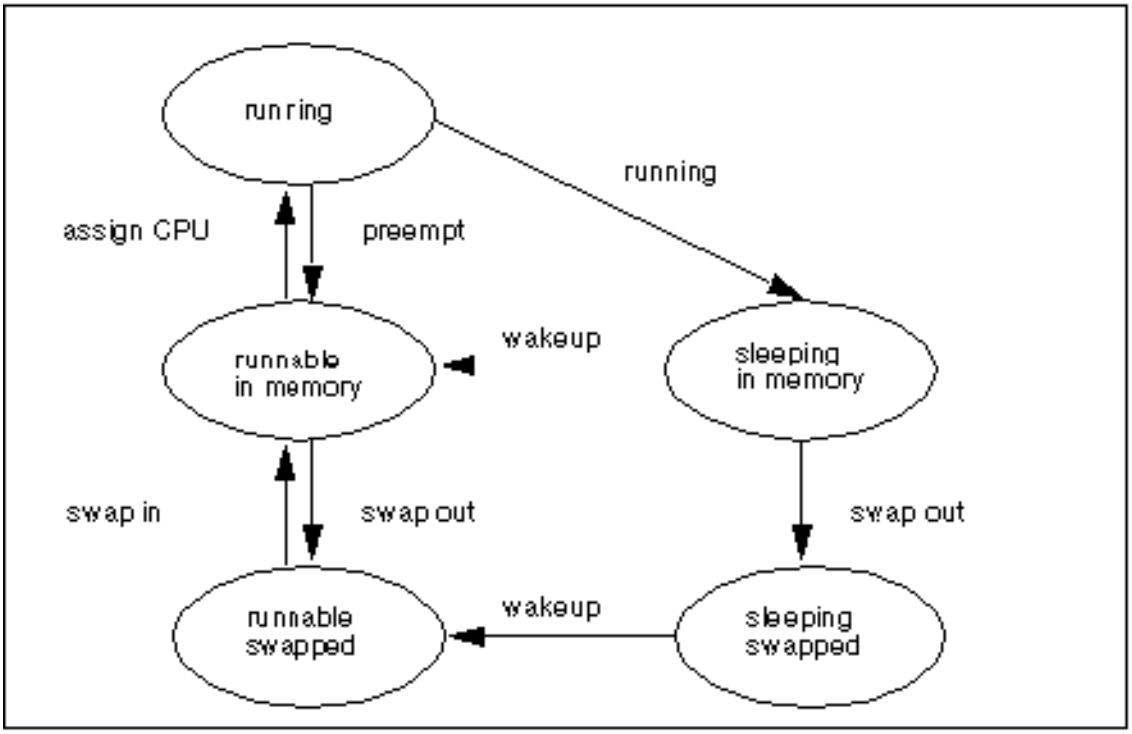


Figure 3: Process state transition diagram

An active process is normally in one of the five states in the diagram. The arrows show how it changes states.

- A process is running if it is assigned to a CPU. A process is preempted—that is, removed from the running state—by the scheduler if a process with a higher priority becomes runnable. A process is also preempted if it consumes its entire time slice and a process of equal priority is runnable.
- A process is runnable in memory if it is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if it is in primary memory but is waiting for a specific event before it can continue execution. For example, a process is sleeping if it is waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, the process is sent a wakeup; if the reason for its sleep is gone, the process becomes runnable.
- A process is runnable and swapped if it is not waiting for a specific event but has had its whole address space written to secondary memory to make room in primary memory for other processes.
- A process is sleeping and swapped if it is both waiting for a specific event and has had its whole address space written to secondary memory to make room in primary memory for other processes.

If a machine does not have enough primary memory to hold all its active processes, it must page or swap some address space to secondary memory:

- When the system is short of primary memory, it writes individual pages of some processes to secondary memory but still leaves those processes runnable. When a process runs, if it accesses those pages, it must sleep while the pages are read back into primary memory.
- When the system gets into a more serious shortage of primary memory, it writes all the pages of some processes to secondary memory and marks those processes as swapped. Such processes get back into a schedulable state only by being chosen by the system scheduler demon process, then read back into memory.

Both paging and swapping, and especially swapping, introduce delay when a process is ready to run again. For processes that have strict timing requirements, this delay can be unacceptable. To avoid swapping delays, real-time processes are never swapped, though parts of them may be paged. An application can prevent paging and swapping by locking its text and data into primary memory. For more information see *memcntl(2)* in [9]. Of course, how much can be locked is limited by how much memory is configured. Also,

locking too much can cause intolerable delays to processes that do not have their text and data locked into memory. Tradeoffs between performance of real-time processes and performance of other processes depend on local needs. On some systems, process locking may be required to guarantee the necessary real-time response.

2.3.4.2 Software latencies

Designers of some real-time applications must have information on software latencies to analyze the performance characteristics of their applications and to predict whether performance constraints can be met. These latencies depend on kernel implementation and on system hardware, so it is not practical to list the latencies. It is useful, however, to describe some of the most important latencies. Consider the following time-line:

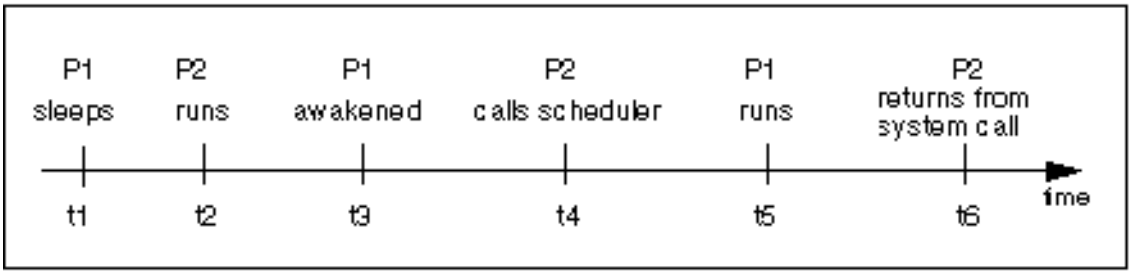


Figure 4: Software latencies

P1 and *P2* represent processes; *t1* through *t6* represent points in time. Suppose that *P1* has a higher priority than all other active processes, including *P2*. *P1* runs and does a system call that causes it to sleep at time *t1*, waiting for I/O. *P2* runs. The I/O device interrupts, resulting in a wakeup at time *t3* that makes *P1* runnable. If *P2* is running in user mode at time *t3*, it is preempted immediately and the interval (*t4 – t3*) is, for practical purposes, zero. If *P2* is running in kernel mode at time *t3*, it is preempted as soon as it gets to a kernel preemption point, a point in kernel code where data structures are in a consistent state and where the state of the current process (*P2* in this example) may be saved and a different process run. Therefore, if *P2* is running in kernel mode at time *t3*, the interval (*t4 – t3*) depends on kernel preemption points, which are spread throughout the kernel. It is useful to know both a typical time to preemption and a maximum time to preemption; these times depend on kernel implementation and on hardware. Eventually, the scheduler runs (at time *t4*), finds that a higher-priority process *P1* is runnable, and runs it. We refer to the interval (*t5 – t4*) as the software switch latency of the system. This latency is, for practical purposes, a constant; again it is an implementation-dependent value. At time *t6*, *P1* returns to the user program from the system call that put it to sleep at time *t1*. For simplicity, suppose that the program is getting only a few bytes of data from the I/O device. In this simple case, the interval (*t6 – t5*) consists basically of the overhead of getting out of the system call. We refer to the interval (*t6 – t3*) as the software wakeup latency of the system; this is the interval from the I/O device interrupt until the user process returns to application level to deal with the interrupt (assuming that it is the highest priority process). So the software wakeup latency is composed of a preemption latency, context-switch time, and a part of system call overhead. Of course, the latency increases as the system call asks for more data.

This discussion of latencies assumes that the text and data of the processes are in primary memory. An application may have to use process locking to guarantee that its processes do not get swapped or paged out of primary memory. See the discussion in the previous section.

2.3.4.3 Primary memory for real-time u-blocks

A process u-block contains per-process information that is not needed when the process is swapped out. The u-block is contained in the *user* structure defined in *user.h*. Normally u-blocks themselves may be swapped or paged. To guarantee software latencies, however, the Reliant UNIX kernel always keeps the u-blocks of real-time processes in primary memory—it never swaps them. (Time sharing u-blocks may be swapped.) Designers of real-time applications should realize that each real-time process has a 6 Kbyte u-area always in primary memory.

2.4 Signals

A signal is an asynchronous notification of an event. The system defines a set of signals that may be delivered to a process.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a "core" image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

Process signals require two special functions:

- the *kill* function, which sends signals,
- the *sigaction* function, which governs how a signal is handled.

A signal is said to be generated for (or sent to) a process when the event associated with that signal first occurs. Examples of such events include hardware faults, timer expiration and terminal activity, as well as the invocation of the *kill* function (see *kill*(2) in [9]). In some circumstances, the same event generates signals for multiple processes.

There are two types of signal: externally generated signals (such as an interrupt from a terminal) and internally generated signals (process errors). Both types are treated in the same way. A signal may be generated in different ways. Examples include:

- a user phase trying to write to protected memory.
- an error in a system call.
- a condition (break or hangup) from the controlling terminal of a process.
- the *kill* system call.
- expiration of the alarm clock timer or a process monitoring trap.

Signals interrupt the normal flow of control in a process. The signals have no direct impact on the execution of a process. They expect the process to provide its own response. Every process has defined actions to be taken in response to signals.

A signal is said to be delivered to a process when the process receives the signal and takes the appropriate action.

2.4.1 Signal types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <*signal.h*>.

Hardware signals are derived from exceptional conditions (faults/errors) which may occur during execution. Such signals include *SIGFPE* representing floating point and other arithmetic exceptions, *SIGILL* for illegal instruction execution, *SIGSEGV* for addresses outside the currently assigned area of memory or for accesses that violate memory protection constraints and *SIGBUS* for accesses that result in hardware related errors. Other, more CPU-specific hardware signals exist, such as *SIGABRT*, *SIGEMT*, and *SIGTRAP*.

Software signals are triggered by user requests: *SIGINT* for the normal interrupt signal; *SIGQUIT* for the more powerful quit signal, that normally causes a core image to be generated; *SIGHUP* and *SIGTERM* that cause graceful process termination, either because a user has hung up, or by user or program request; and *SIGKILL*, a more powerful termination signal which a process cannot catch or ignore. Programs may define their own asynchronous events using *SIGUSR1* and *SIGUSR2*. Other software signals (*SIGALRM*, *SIGVTALRM*, *SIGPROF*) indicate the expiration of interval timers.

A process can request notification via a *SIGPOLL* signal when input or output is pending for a descriptor, or when a "non-blocking" operation completes. A process may request to receive a *SIGURG* signal when an urgent condition arises.

A process may be "stopped" by a signal sent to it or the members of its process group. The *SIGSTOP* signal is a powerful stop signal, because it cannot be caught. Other stop signals *SIGTSTP*, *SIGTTIN*, and *SIGTTOU* are used when a user request, input request, or output request respectively is the reason for stopping the

process. A *SIGCONT* signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a *SIGCHLD* signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. *SIGXCPU* occurs when a process nears its CPU time limit and *SIGXFSZ* warns that the limit on file size creation has been reached.

2.4.2 Signal handlers

The *<signal.h>* header file defines a default action for each signal. This action is selected from the following:

Abort
: On receiving this signal the receiving process is abnormally terminated with all the consequences described under *exit*.

Exit
: On receiving this process the receiving process is normally terminated with all the consequnces described under *exit*.

Stop
: On receiving this signal the receiving process is to stop

Ignore
: On receiving this signal the receiving process is to ignore it.

As the default action in response to a signal is process termination, a process which needs to change the default response and define its own signal handling has to use *sigaction*. The *sigaction* function takes three arguments:

- the first argument specifies the signal.
- the second argument specifies the signal handling action.
- the third argument returns the old action.

The first argument is an integer value standing for a signal. The second and third arguments select one of three types of response that can be defined for a signal:

1. Use the default response to the signal - *SIG_DFL*
2. Ignore the signal - *SIG_IGN*
3. Catch the signal by calling a function - *pointer to an action*

The *<signal.h>* header file defines the particular values used to request that a default action (*SIG_DFL*) should be taken in response to the signal or that the signal should be ignored (*SIG_IGN*), and it also defines the *sigaction* structure used to specify signal handling functions. The second and third arguments of the *sigaction* function are pointers to the *sigaction* structure defined in the *<signal.h>* header file. The *<signal.h>* header file further defines symbolic names for the signal numbers. It must always be included when signals are to be used.

To control the way a signal is delivered, a process calls *sigaction*, thereby assigning a handler to that signal. The call

```
#include <signal.h>
sigaction(signo, sa, osa)
      int signo;
      struct sigaction *sa;
      struct sigaction *osa;
```

assigns interrupt handler address *sa_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants *SIG_IGN* and *SIG_DFL* used as values for *sa_handler* cause ignoring or defaulting of a condition.

There are two things that must be done within a signal handler. Resetting the routine that catches the signal

[signal(n, SIG_DFL);]

is only the first. It is also necessary to unblock the blocked signal, which is done with *sigprocmask*.

sa_mask specifies the set of signals to be masked when the handler is invoked; it implicitly includes the signal which invoked the handler. Five operations are permitted on signal sets. The set will be emptied by a call to *sigemptyset*. It will be filled with every signal currently supported by a call to *sigfillset*. Specific signals may be added or deleted with calls to *sigaddset* and *sigdelset* respectively. Set membership can be tested with *sigismember*. Signals sets should always be initialized with a call to *sigemptyset* or *sigfillset*.

sa_flags specifies special properties of the signal, such as whether system calls should be restarted if the signal handler returns, if the signal action should be reset to *SIG_DFL* when it is caught, and whether the handler should operate on the normal run-time stack or a special signal stack (see below).

If *osa* is nonzero, the previous signal action is passed in *osa*.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sa_mask* to a set of those masked for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The signal handling routine *sa_handler* is called by a C call of the form

(*sa_handler)(int□signo,□siginfo_t□*infop,□ucontext_t□*ucp);

signo is the number of the signal that occurred. *infop* is either equal to 0, or points to a structure that contains information detailing the reason why the signal was generated. This information must be explicitly asked for when the signal's action is specified. The *ucp* parameter is a pointer to a structure containing the process's context prior to the delivery of the signal, and will be used to restore the process's context upon return from the signal handler.

In the following example the first call to *sigaction* causes interrupts to be ignored. The second call to *sigaction* reinstates the default response to interrupts (process termination).

```
main()□{
□□□#include□<signal.h>
□□□struct□sigaction□new_act,□old_act;
□□□new_act.sa_handler□=□SIG_IGN;
□□□sigaction(SIGINT,□&new_act,□&old_act);
□□□new_act.sa_handler□=□SIG_DFL;
□□□sigaction(SIGINT,□&new_act,□&old_act);
}
```

In both cases *sigaction* returns the old action in the last argument *old_act*.

First, even before the *main* function starts, all signals are set to *SIG_DFL* or *SIG_IGN* (see *exec(2)* in [9]). If an action is defined for a specific signal, it is normally retained until some other action is defined explicitly by an invocation of *signal*, *sigset*, *sigignore* or *sigaction* or until the process is executed (*exec*) (see *signal(2)*, *sigset(2)* and *sigaction(2)* and also *exec(2)* in [9]). When a process is executed (*exec*), all signals set to be caught are automatically reset to *SIG_DFL*. A process may also ask for the action for a signal to be automatically reset to *SIG_DFL* once the signal has been caught (see *signal(2)* and *sigaction(2)*).

Instead of the values *SIG_IGN* or *SIG_DFL*, the second argument to *sigaction* may also specify a signal handling function. In such cases the specified function is called when the signal is received. This is mostly done to allow the program to complete unfinished operations (such as deleting a temporary file) before terminating. For example:

```
#include□<signal.h>
main()□{
□□□struct□sigaction□new_act,□old_act;
□□□void□on_intr();
□□□new_act.sa_handler□=□SIG_IGN;
```

```
□□□sigaction(SIGINT,□&new_act,□&old_act);
□□□if□(old_act.sa_handler□!=□SIG_IGN)□{
□□□□□□new_act.sa_handler□=□on_intr;
□□□□□□sigaction(SIGINT,□&new_act,□&old_act);
□□□}
□□□/*□do□processing□*/
□□□exit(0);
}
void□on_intr()□{
□□□unlink(tempfile);
□□□exit(1);
}
```

Before *on_intr* is defined as a signal handling routine for the *SIGINT* interrupt signal, the program checks the interrupt handling status. Interrupts continue to be ignored if they were already being ignored. This is essential because signals such as interrupt signals are sent to all processes started from a given terminal. If there is a non-interactive program running as a background process (started with *&*), the shell disables interrupts for it so that it is not afffected by interrupts intended for foreground processes. If this program were to define at the outset that all interrupts were to be caught by the *on_intr* function, this would override the shell's efforts to protect the program from interrupts as a background process.

The solution shown above is first to call *sigaction* for *SIGINT* in order to query the current response to the interrupt signal. This is returned in the third argument of *sigaction*. If interrupts are already being ignored, the process is to continue ignoring them. If not, they are to be caught. In this case the second invocation of *sigaction* for *SIGINT* defines a new response to the signal, the signal handling routine specified by *on_intr*.

2.4.3 Sending signals

A process may receive a signal from another process, the terminal or the system. For most signals the process can define whether it will terminate on receiving the signal, completely ignore it, or catch it and respond to it in a manner defined by the user process. Thus an *INTERRUPT* signal might be sent as a result of a key being pressed on the terminal keyboard (*delete*, *break* or *rubout*). The response to this event depends on the requirements of the program currently executing. For example:

- The shell runs most commands in such a way that they stop executing immediately on receiving an interrupt. This typically applies, for example, to the *pr* command. This alows the user to suppress unrequired output.
- The shell itself ignores interrupts when reading from the terminal because it has to continue executing even if the user terminates a command such as *pr*.
- The *ed* editor catches interrupts so that it can terminate an operation (such as output) with itself being terminated.

Each signal type is assigned a different integer value. Thus a value of *1* stands for a hangup signal. The signal number acts as a subscript to the signal array of the receiving process. For each signal type the signal arraay contains the address of a signal handling function defined in the user process. If there is no function defined, *0* or *1* is used. If the value is *1*, the signal is ignored. If it is *0*, the default action is taken.

A child process inherits its default actions from its parent and ignores signals. Caught signals produce the child process' default action. This has to be so because the address data for signal handling functions specified in the parent process does not apply to the child.

A process can send a signal to another process or group of processes with the calls:

```
kill(int pid, int signo);
sigsend(idetype_t idtype, id_t id, int signo);
```

Unless the process sending the signal is privileged, its real or effective user ID must be equal to the receiving process's real or saved user ID.

Signals can also be sent from from a terminal device to the process group or session leader associated with the terminal. See *termio(7)*.

2.4.4 Protecting critical sections

To block a section of code against one or more signals, a *sigprocmask* call may be used to add a set of signals to the existing mask, and return the old mask:

sigprocmask(SIG_BLOCK, sigset_t *mask, sigset_t *omask);

The old mask can then be restored later with *sigprocmask*:

sigprocmask(SIG_SETMASK, sigset_t *omask, sigset_t *mask);

The *sigprocmask* call can be used to read the current mask without changing it by specifying a null pointer as its second argument.

The following call is used to pause waiting for a signal which either invokes a handling routine or terminates the process.

sigsuspend(sigset_t *mask);

2.4.5 Signal stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigaltstack {
    caddr_t ss_sp;
    int     ss_size;
    int     ss_flags;
};
sigaltstack(ss, oss)
    struct sigaltstack *ss;
    struct sigaltstack *oss;
```

to provide the system with a stack based at *ss_sp* of size *ss_size* for delivery of signals. The system automatically adjusts for direction of stack growth. *ss_flags* indicates whether the process is currently on the signal stack, and whether the signal stack is disabled.

When a signal is to be delivered and the process has requested that it be delivered on the alternate stack (see *sigaction* above), the system checks whether the process is on a signal stack. If it is not, then the process is switched to the alternate stack, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a nonlocal exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigaltstack* call should be used to reset the signal stack.

2.5 Interprocess communication

Reliant UNIX 5.43 provides several mechanisms that allow processes to exchange data and synchronize execution. The simpler of these mechanisms are pipes, named pipes, and signals. These are limited, however, in what they can do. For instance,

- Pipes do not allow unrelated processes to communicate.
- Named pipes allow unrelated processes to communicate, but they cannot provide private channels for pairs of communicating processes; that is, any process with appropriate permission may read from or write to a named pipe.
- Sending signals, via the *kill* system call, allows arbitrary processes to communicate, but the message consists only of the signal number.

Reliant UNIX 5.43 also provides an InterProcess Communication (IPC) package that supports three, more versatile types of interprocess communication. For example,

- Messages allow processes to send formatted data streams to arbitrary processes.
- Semaphores allow processes to synchronize execution.
- Shared memory allows processes to share parts of their virtual address space.

When implemented as a unit, these three mechanisms share common properties such as

- each mechanism contains a "get" system call to create a new entry or retrieve an existing one

- each mechanism contains a "control" system call to query the status of an entry, to set status information, or to remove the entry from the system
- each mechanism contains an "operations" system call to perform various operations on an entry

This chapter describes the system calls for each of these three forms of IPC.

This information is for programmers who write multiprocess applications. These programmers should have a general understanding of what semaphores are and how they are used.

Information from other sources would also be helpful. See the *ipcs(1)* and *ipcrm(1)* manual pages in [4] and the following manual pages in [9]:

intro(2), lvlipc(2), msgget(2), msgctl(2), msgop(2), semget(2), semctl(2), semop(2), shmget(2), shmctl(2), shmop(2)

Included in this chapter are several example programs that show the use of these IPC system calls. Since there are many ways to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function provided by the calls.

2.5.1 Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can send and receive messages.

Before a process can send or receive a message, it must have the Reliant UNIX operating system generate the necessary software mechanisms to handle these operations. A process does this using the *msgget* system call. In doing this, the process becomes the owner/creator of a message queue and specifies the initial operation permissions for all processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the *msgctl* system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use *msgctl* to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process which is attempting to send a message can wait until it becomes possible to post the message to the specified message queue; the receiving process isn't involved (except indirectly, e.g., if the consumer isn't consuming, the queue space will eventually be exhausted) and vice versa. A process which specifies that execution is to be suspended is performing a " blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

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

1. It is successful.
2. It receives a signal.
3. The message queue is removed from the system.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (–1) is returned to the process, and an external error number variable, *errno*, is set accordingly.

2.5.1.1 Using messages

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier is called the message queue identifier (*msqid*); it is used to identify or refer to the associated message queue and data structure.

The message queue is used to store (header) information about each message being sent or received. This information, which is for internal use by the system, includes the following for each message:

- pointer to the next message on queue
- message type

- message text size
- message text address

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- pointer to first message on the queue
- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

Include files discussed in this section are located in the */usr/include* or */usr/include/sys* directories.

The structure definition for the associated data structure is as follows:

```
struct msqid_ds
{
      struct ipc_perm  msg_perm;     /* operation permission struct */
      struct msg       *msg_first;   /* ptr to first message on q */
      struct msg       *msg_last;    /* ptr to last message on q */
      ulong            msg_cbytes;   /* current # bytes on q */
      ulong            msg_qnum;     /* # of messages on q */
      ulong            msg_qbytes;   /* max # of bytes on q */
      pid_t            msg_lspid;    /* pid of last msgsnd */
      pid_t            msg_lrpid;    /* pid of last msgrcv */
      time_t           msg_stime;    /* last msgsnd time */
      long             msg_pad1;     /* reserved for time_t expansion */
      time_t           msg_rtime;    /* last msgrcv time */
      long             msg_pad2;     /* time_t expansion */
      time_t           msg_ctime;    /* last change time */
      long             msg_pad3;     /* time expansion */
      long             msg_pad4[4];  /* reserve area*/
 };
```

It is located in the <*sys/msg.h*> header file. Note that the *msg_perm* member of this structure uses *ipc_perm* as a template. The following figure shows the breakout for the operation permissions data structure.

The definition of the *ipc_perm* data structure is as follows:

```
struct ipc_perm
{
     uid_t uid;     /* owner's user id */
     gid_t gid;     /* owner's group id */
     uid_t cuid;    /* creator's user id */
     gid_t cgid;    /* creator's group id */
     mode_t mode;   /* access modes */
     ulong seq;     /* slot usage sequence number */
     key_t key;     /* key */
     long pad[4];   /*reserve area */
 };
```

It is located in the <*sys/ipc.h*> header file and is common to all IPC facilities.

The *msgget* system call is used to perform one of two tasks:

- to get a new message queue identifier and create an associated message queue and data structure for it
- to return an existing message queue identifier that already has an associated message queue and data structure

Both tasks require a *key* argument passed to the *msgget* system call. For the first task, if the *key* is not already in use for an existing message queue identifier at the security level of the calling process, a new identifier is returned with an associated message queue and data structure created for the *key*. The new *msqid* inherits the security level of the creating process. This occurs as long as no system-tunable parameters would be exceeded and a control command *IPC_CREAT* is specified in the *msgflg* argument passed in the system call.

There is also a provision for specifying a *key* of value zero, known as the private key (*IPC_PRIVATE*). When specified, a new identifier at the security level of the creating process is always returned with an associated message queue and data structure created for it unless a system-tunable parameter would be exceeded.

The *ipcs* command will show the *KEY* field for the *msqid* as all zeros.

For the second task, if a message queue identifier exists at the security level of the calling process for the *key* specified, the value of the existing identifier is returned. If you do not want to have an existing message queue identifier returned, a control command (*IPC_EXCL*) can be specified (set) in the *msgflg* argument passed to the system call. ("Using *msgget*" describes how to use this system call.)

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. The message queue creator also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, *msgop* (message operations) and *msgctl* (message control) can be used.

Message operations, as mentioned before, consist of sending and receiving messages. The *msgsnd* and *msgrcv* system calls are provided for each of these operations (see Section "Operations for messages", for details of these system calls).

The *msgctl* system call permits you to control the message facility in the following ways:

- by retrieving the data structure associated with a message queue identifier (*IPC_STAT*)
- by changing operation permissions for a message queue (*IPC_SET*)
- by changing the size (*msg_qbytes*) of the message queue for a particular message queue identifier (*IPC_SET*)
- by removing a particular message queue identifier from the Reliant UNIX operating system along with its associated message queue and data structure (*IPC_RMID*)

See Section "Controlling message queues", for *msgctl* system call details.

2.5.1.2 Getting message queues

This section describes how to use the *msgget* system call. The accompanying program illustrates its use.

Using msgget

The synopsis found in the *msgget(2)* entry in [9] is as follows:

```
#include□□<sys/types.h>
#include□□<sys/ipc.h>
#include□□<sys/msg.h>
int□□msgget□(key,□msgflg)
key_t□□key;
int□msgflg;
```

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

The following line in the synopsis:

int msgget (key, msgflg)

informs you that *msgget* is a function with two formal arguments that returns an integer-type value. The next two lines:

key_t□□key;
int□msgflg;

declare the types of the formal arguments. *key_t* is defined by a *typedef* in the <*sys/types.h*> header file to be an integral type.

The integer returned from this function upon successful completion is the message queue identifier that was discussed earlier. Upon failure, the external variable *errno* is set to indicate the reason for failure, and the value -1 (which is not a valid *msqid*) is returned.

As declared, the process calling the *msgget* system call must supply two arguments to be passed to the formal *key* and *msgflg* arguments.

A new *msqid* with an associated message queue and data structure is provided if either

- *key* is equal to *IPC_PRIVATE*, or
- *key* is a unique integer and the control command *IPC_CREAT* is specified in the *msgflg* argument.

The value passed to the *msgflg* argument must be an integer-type value that will specify the following:

- operations permissions
- control fields (commands)

Operation permissions determine the operations that processes are permitted to perform on the associated message queue. "Read" permission is necessary for receiving messages or for determining queue status by means of a *msgctl IPC_STAT* operation. "Write" permission is necessary for sending messages. The Table 17 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation permissions	Octal value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Table 17: Operation permission codes

A specific value is derived by adding or bitwise ORing the octal values for the operation permissions wanted. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the <*sys/msg.h*> header file which can be used for the user operations permissions. They are as follows:

MSG_W 0200 /* write permissions by owner */
MSG_R 0400 /* read permissions by owner */

Control flags are predefined constants (represented by all uppercase letters). The flags which apply to the *msgget* system call are *IPC_CREAT* and *IPC_EXCL* and are defined in the <*sys/ipc.h*> header file.

The value for *msgflg* is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by adding or bitwise ORing (-|-) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *msgflg* value can easily be set by using the flag names in conjunction with the octal operation permissions value:

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

As specified by the *msgget(2)* page in [9], success or failure of this system call depends upon the argument values for *key* and *msgflg* or system-tunable parameters. The system call will attempt to return a new message queue identifier if one of the following conditions is true:

- *key* is equal to *IPC_PRIVATE*
- *key* does not already have a message queue identifier associated with it is not yet associated with a message queue identifier at the security level of the creating process and (*msgflg* and *IPC_CREAT*) is "true" (not zero).

The *key* argument can be set to *IPC_PRIVATE* like this:

msqid = msgget (IPC_PRIVATE, msgflg);

The system call will always be attempted. Exceeding the *MSGMNI* system-tunable parameter always causes a failure. The *MSGMNI* system-tunable parameter determines the systemwide number of unique message queues that may be in use at any given time.

IPC_EXCL is another control command used in conjunction with *IPC_CREAT*. It will cause the system call to return an error if a message queue identifier already exists at the security level of the calling process for the specified *key*. This is necessary to prevent the process from thinking that it has received a new identifier when it has not. In other words, when both *IPC_CREAT* and *IPC_EXCL* are specified, a new message queue identifier is returned if the system call is successful.

Refer to the *msgget(2)* page in [9] for specific, associated data structure initialization for successful completion. The specific failure conditions and their error names are contained there also.

Example program

The figure "msgget system call example" is a menu-driven program. It allows all possible combinations of using the msgget system call to be exercised.

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

This program begins (lines 4-8) by including the required header files as specified by the *msgget(2)* entry in [9]. Note that the <*sys/errno.h*> header file is included as opposed to declaring *errno* as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self explanatory. These names make the programs more readable are perfectly legal since they are local to the program. The variables declared for this program and what they are used for are as follows:

key
: used to pass the value for the desired "key"

opperm
: used to store the desired operation permissions

flags
: used to store the desired control commands (flags)

opperm_flags
: used to store the combination from the logical ORing of the *opperm* and *flags* variables; it is then used in the system call to pass the *msgflg* argument

msqid
: used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal key, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows errors to be observed 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 (line 57).

If no error occurred, the returned message queue identifier is displayed (line 61).

The example program for the *msgget* system call follows. We suggest you name the program file *msgget.c* and the executable file *msgget*.

msgget system call example

```
□1□□□□/*□This□is□a□program□to□illustrate
□2□□□□**□the□message□get,□msgget(),
□3□□□□**□system□call□capabilities.□*/
□4□□□□#include□□□□<stdio.h>
□5□□□□#include□□□□<sys/types.h>
□6□□□□#include□□□□<sys/ipc.h>
□7□□□□#include□□□□<sys/msg.h>
□8□□□□#include□□□□<errno.h>
□9□□□□/*□Start□of□main□C□language□program□*/
10□□□□main()
11□□□□{
12□□□□□□□□key_t□key;
13□□□□□□□□int□opperm,□flags;
14□□□□□□□□int□msqid,□opperm_flags;
15□□□□□□□□/*□Enter□the□desired□key□*/
16□□□□□□□□printf("Enter□the□desired□key□in□hex□=□");
17□□□□□□□□scanf("%x",□&key);
18□□□□□□□□/*□Enter□the□desired□octal□operation
19□□□□□□□□□□□□permissions.□*/
20□□□□□□□□printf("\nEnter□the□operation\n");
21□□□□□□□□printf("permissions□in□octal□=□");
22□□□□□□□□scanf("%o",□&opperm);
23□□□□□□□□/*□Set□the□desired□flags.□*/
24□□□□□□□□printf("\nEnter□corresponding□number□to\n");
25□□□□□□□□printf("set□the□desired□flags:\n");
26□□□□□□□□printf("No□flags□□□□□□□□□□□□□□□□□□□□□=□0\n");
27□□□□□□□□printf("IPC_CREAT□□□□□□□□□□□□□□□□□□□=□1\n");
28□□□□□□□□printf("IPC_EXCL□□□□□□□□□□□□□□□□□□□=□2\n");
29□□□□□□□□printf("IPC_CREAT□and□IPC_EXCL□□□□=□3\n");
30□□□□□□□□printf("□□□□□□□□□□□□□Flags□□□□□□□□□□=□");
31□□□□□□□□/*□Get□the□flag(s)□to□be□set.□*/
32□□□□□□□□scanf("%d",□&flags);
33□□□□□□□□/*□Check□the□values.□*/
34□□□□□□□□printf□("\nkey□=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□is□unsuccessful.□*/
55□□□□□□□□□if(msqid□==□-1)
56□□□□□□□□□{
57□□□□□□□□□printf□("\nThe□msgget□call□failed,□error□number□=□%d\n",□errno);
58□□□□□□□□□}
59□□□□□□□□□/*□Return□the□msqid□upon□successful□completion.□*/
60□□□□□□□□□else
61□□□□□□□□□□□□□□printf□("\nThe□msqid□=□%d\n",□msqid);
62□□□□□□□□□exit(0);
63□□□□}
```

2.5.1.3 Controlling message queues

This section describes how to use the *msgctl* system call. The accompanying program illustrates its use.

Using msgctl

The synopsis found in the *msgctl(2)* entry in [9] 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; it returns an integer-type value.

When successful, it returns a zero value; 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 any one of the following values:

IPC_STAT
: return the status information contained in the associated data structure for the specified message queue identifier, and place it in the data structure pointed to by the *buf* pointer in the user memory area.

IPC_SET
: for the specified message queue identifier, set the effective user and group identification, operation permissions, and the number of bytes for the message queue to the values contained in the data structure pointed to by the *buf* pointer in the user memory area.

IPC_RMID
: remove the specified message queue identifier along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or system administrator 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 following example program. If you need more information on the logic manipulations in this program, read the *msgget(2)* section of [9]; it goes into more detail than would be practical for this document.

Example program

The figure "msgctl system call example" is a menu-driven program. It allows all possible combinations of using the *msgctl* system call to be exercised.

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

This program begins (lines 5-9) by including the required header files as specified by the *msgctl(2)* entry in [9]. Note in this program that *errno* is declared as an external variable, and therefore, the <*sys/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 are perfectly legal since they are local to the program. The variables declared for this program and what they are used for are as follows:

uid
: used to store the *IPC_SET* value for the effective user identification

gid
: used to store the *IPC_SET* value for the effective group identification

mode
: used to store the *IPC_SET* value for the operation permissions

bytes
: used to store the *IPC_SET* value for the number of bytes in the message queue (*msg_qbytes*)

rtrn
: used to store the return integer value from the system call

msqid
: used to store and pass the message queue identifier to the system call

command
: used to store the code for the desired control command so that subsequent processing can be performed on it

choice
: used to determine which member is to be changed for the *IPC_SET* control command

msqid_ds
: used to receive the specified message queue identifier's data structure when an *IPC_STAT* control command is performed

buf
: a pointer passed to the system call which locates the data structure in the user memory area where the *IPC_STAT* control command is to place its return values or where the *IPC_SET* command gets the values to set

Note that the *msqid_ds* data structure in this program (line 16) uses the data structure, located in the <*sys/msg.h*> header file of the same name, as a template for its declaration.

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 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 (line 108). If the system call is successful, a message indicates this along with the message queue identifier used (lines 110-113).

If the *IPC_SET* control command is selected (code 2), the first thing 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 into 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 Reliant UNIX operating system. Note that the *buf* pointer is ignored in performing 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. We suggest that you name the source program file *msgctl.c* and the executable file *msgctl*.

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

```
28□□□□□□□□□/*□Check□the□values.□*/
29□□□□□□□□□printf□("\nmsqid□=%d,□command□=□%d\n",
30□□□□□□□□□□□□□msqid,□command);
31□□□□□□□□□switch□(command)
32□□□□□□□□□{
33□□□□□□□□□case□1:□□□□/*□Use□msgctl()□to□duplicate
34□□□□□□□□□□□□□□□□□□□□□□□□the□data□structure□for
35□□□□□□□□□□□□□□□□□□□□□□□□msqid□in□the□msqid_ds□area□pointed
36□□□□□□□□□□□□□□□□□□□□□□□□to□by□buf□and□then□print□it□out.□*/
37□□□□□□□□□□□□□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;□□□□□4
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
62□□□□□□□□□□□□□□□□pass□as□an□illegal□entry□will
63□□□□□□□□□□□□□□□□cause□repetitive□failures□until
64□□□□□□□□□□□□□□□□msqid_ds□is□updated□with
65□□□□□□□□□□□□□□□□IPC_STAT.□*/
66□□□□□□□□□□□□□switch(choice){
67□□□□□□□□□□□□□case□1:
68□□□□□□□□□□□□□□□□printf("\nEnter□USER□ID□=□");
69□□□□□□□□□□□□□□□□scanf□("%ld",□&uid);
70□□□□□□□□□□□□□□□□buf–>msg_perm.uid□=(uid_t)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,□(struct□msqid_ds□*)□NULL);
104□□□□□□□□□}
105□□□□□□□□□/*□Perform□the□following□if□the□call□is□unsuccessful.□*/
106□□□□□□□□□if(rtrn□==□–1)
107□□□□□□□□□{
108□□□□□□□□□□□printf□("\nThe□msgctl□call□failed,□error□number□=□%d\n",\□errno);
109□□□□□□□□□}
110□□□□□□□□□/*□Return□the□msqid□upon□successful□completion.□*/
111□□□□□□□□□else
112□□□□□□□□□□□□□□□printf□("\nMsgctl□was□successful□for□msqid□=□%d\n",
113□□□□□□□□□□□□□□□□□□msqid);
114□□□□□□□□□exit□(0);
115□□□□}
```

2.5.1.4 Operations for messages

This section describes how to use the *msgsnd* and *msgrcv* system calls. The accompanying program illustrates their use.

Using msgop

The synopsis found in the *msgop(2)* entry in [9] is as follows:

```
#include□<sys/types.h>
#include□<sys/ipc.h>
#include□<sys/msg.h>
int□msgsnd□(msqid,□msgp,□msgsz,□msgflg)
int□msqid;
struct□msgbuf□*msgp;
int□msgsz,□msgflg;
int□msgrcv□(msqid,□msgp,□msgsz,□msgtyp,□msgflg)
int□msqid;
struct□msgbuf□*msgp;
int□msgsz;
long□msgtyp;
int□msgflg;
```

Sending a message

The *msgsnd* system call requires four arguments to be passed to it. It returns an integer value.

When successful, it returns a zero value; 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 *msgflg* argument allows the "blocking message operation" to be performed if the *IPC_NOWAIT* flag is not set ((*msgflg* AND *IPC_NOWAIT*)= = 0); the operation would block if the total number of bytes allowed on the specified message queue are in use (*msg_qbytes* OR *MSGMNB*), or the total system-wide number of messages on all queues is equal to the system-imposed limit (*MSGTQL*). If the *IPC_NOWAIT* flag is set, the system call will fail and return a −1.

The *msg_qbytes* data structure member can be lowered from *MSGMNB* by using the *msgctl IPC_SET* control command, but only the system administrator can raise it afterwards.

Further details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, read Using msgget. It goes into more detail than would be practical for every system call.

Receiving messages

The *msgrcv* system call requires five arguments to be passed to it; it returns an integer value.

When successful, it returns a value equal to the number of bytes received; when unsuccessful it returns a −1.

The *msqid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the *msgget* system call.

The *msgp* argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The *msgsz* argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired (see the *msgflg* argument below).

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* AND *IPC_NOWAIT*) == 0); the operation would block if there is not a message on the message queue of the desired type (*msgtyp*) to be received. If the *IPC_NOWAIT* flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. *msgflg* can also specify that the system call fail if the message is longer than the "size" to be received; this is done by not setting the *MSG_NOERROR* flag in the *msgflg* argument ((*msgflg* AND *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 following program. If you need more information on the logic manipulations in this program read Using msgget". It goes into more detail than would be practical for every system call.

Example program

The figure "msgop system call example" is a menu-driven program. It allows all possible combinations of using the *msgsnd* and *msgrcv* system calls to be exercised.

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

This program begins (lines 5-9) by including the required header files as specified by the *msgop(2)* entry in [9]. Note that in this program *errno* is declared as an external variable; therefore, the <*sys/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 are perfectly legal since they are local to the program. The variables declared for this program and what they are used for are as follows:

sndbuf
: used as a buffer to contain a message to be sent (line 13); it uses the *msgbuf1* data structure as a template (lines 10-13). The *msgbuf1* structure (lines 10-13) is a duplicate of the *msgbuf* structure contained in the <*sys/msg.h*> header file, except that the size of the character array for *mtext* is tailored to fit this application. The *msgbuf* structure should not be used directly because *mtext* has only one element that would limit the size of each message to one character. Instead, declare your own structure. It should be identical to *msgbuf* except that the size of the *mtext* array should fit your application.

rcvbuf
: used as a buffer to receive a message (line 13); it uses the *msgbuf1* data structure as a template (lines 10-13)

msgp
: used as a pointer (line 13) to both the *sndbuf* and *rcvbuf* buffers

i
: used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the *msgsnd* system call; it is also used as a counter to output the received message for the *msgrcv* system call

c
: used to receive the input character from the *getchar* function (line 50)

flag
: used to store the code of *IPC_NOWAIT* for the *msgsnd* system call (line 61)

flags
: used to store the code of the *IPC_NOWAIT* or *MSG_NOERROR* flags for the *msgrcv* system call (line 117)

choice
: used to store the code for sending or receiving (line 30)

rtrn
: used to store the return values from all system calls

msqid
: used to store and pass the desired message queue identifier for both system calls

msgsz
: used to store and pass the *size* of the message to be sent or received

msgflg
: used to pass the value of *flag* for sending or the value of *flags* for receiving

msgtyp
: used for specifying the message type for sending or for picking a message type for receiving.

Note that a *msqid_ds* data structure is set up in the program (line 21) with a pointer initialized to point to it (line 22); this will allow the data structure members affected by message operations to be observed. They are observed by using the *msgctl* (*IPC_STAT*) system call to get them for the program to print them out (lines 80-92 and lines 160-167).

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; 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 will continue until an end-of-file is recognized which, for the *getchar* function, is a control-D (CTRL-D) immediately following a carriage return (<CR>).

The message is immediately echoed from the *mtext* array of the *sndbuf* data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the *IPC_NOWAIT* flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored 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 performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed and should be zero (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

When the code is to receive a message, 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; it is stored in *msqid* (lines 100-103).

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

The code for the desired combination of control flags is requested next; it is stored in *flags* (lines 108-117). Depending upon the selected combination, *msgflg* is set accordingly (lines 118-131).

Finally, the number of bytes to be received is requested; it is stored in *msgsz* (lines 132-135).

The *msgrcv* system call is performed (line 142). If it is unsuccessful, a message and error number is displayed (lines 143-145). If successful, a message indicates so, and the number of bytes returned and the *msg* type returned (because the value returned may be different from the value requested) is displayed followed by the received message (lines 150-156).

When a message is successfully received, three members of the associated data structure are updated. They are:

msg_qnum
: contains the number of messages on the message queue; it is decremented by one.

msg_lrpid
: contains the 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.

This figure shows the *msgop* system calls. We suggest that you put the program into a source file called *msgop.c* and then compile it into an executable file called *msgop*.

msgop system call example

```
□1□□□□/*□This□is□a□program□to□illustrate
□2□□□□**□the□message□operations,□msgop(),
□3□□□□**□system□call□capabilities
□4□□□□*/
□5□□□□/*□Include□necessary□header□files□*/
□6□□□□#include□□□□<stdio.h>
□7□□□□#include□□□□<sys/types.h>
□8□□□□#include□□□□<sys/ipc.h>
□9□□□□#include□□□□<sys/msg.h>
10□□□□struct□msgbuf1□{
11□□□□□□□□long□□□□mtype;
12□□□□□□□□char□□□□mtext[8192];
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□of\n"
35□□□□□□□□□□□□□□printf("the□message□queue□to\n")
36□□□□□□□□□□□□□□printf("handle□the□message□=□");
37□□□□□□□□□□□□□□scanf("%d",□&msqid);
38□□□□□□□□□□□□□□/*□Set□the□message□type□*/
39□□□□□□□□□□□□□□printf("\nEnter□a□positive□integer\n"
40□□□□□□□□□□□□□□printf("message□type□(long)□for□the\n")
41□□□□□□□□□□□□□□printf("message□=□");
42□□□□□□□□□□□□□□scanf("%ld",□&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;
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□\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□bufffer□*/
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□message□on□the□queue
```

```
105□□□□□□□□□□□□□□□□□by□using□its□type□*/
106□□□□□□□□□□□□□□printf("\nEnter□the□msgtyp□=□")
107□□□□□□□□□□□□□□scanf("%ld",□&msgtyp);
150□□□□□□□□□□□□□□□□□□/*□Print□the□number□of□bytes□received;
151□□□□□□□□□□□□□□□□□□□□□□it□is□equal□to□the□return
152□□□□□□□□□□□□□□□□□□□□□□value□*/
153□□□□□□□□□□□□□□□□□□printf("Bytes□received□=□%d\n",□rtrn)
154□□□□□□□□□□□□□□□□□□/*□Print□the□received□message□*/
155□□□□□□□□□□□□□□□□□□for(i□=□0;□i<rtrn;□i++)
156□□□□□□□□□□□□□□□□□□□□□□putchar(rcvbuf.mtext[i]);
157□□□□□□□□□□□□□□}
158□□□□□□□□□□□□□□/*□Check□the□associated□data□structure□*/
159□□□□□□□□□□□□□□msgctl(msqid,□IPC_STAT,□buf);
160□□□□□□□□□□□□□□/*□Print□the□decremented□number□of□messages□*/
161□□□□□□□□□□□□□□printf("\nThe□msg_qnum□=□%d\n",□buf->msg_qnum
162□□□□□□□□□□□□□□/*□Print□the□process□id□of□the□last□receiver□*/
163□□□□□□□□□□□□□□printf("The□msg_lrpid□=□%d\n",□buf->msg_lrpid)
164□□□□□□□□□□□□□□/*□Print□the□last□message□receive□time□*/
165□□□□□□□□□□□□□□printf("The□msg_rtime□=□%d\n",□buf->msg_rtime)
166□□□□□□□□□}
167□□□□}
```

2.5.2 Semaphores

The semaphore type of IPC allows processes (executing programs) to communicate through the exchange of semaphore values. Since many applications require the use of more than one semaphore, the Reliant UNIX operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, *SEMMSL*, has a default value of 25. Semaphore sets are created by using the *semget* (semaphore get) system call.

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

Any process can manipulate the semaphore(s) if the owner of the semaphore grants permission. Each semaphore within a set can be incremented and decremented with the *semop(2)* system call (documented in [9]).

To increment a semaphore, an integer value of the desired magnitude is passed to the *semop* system call. To decrement a semaphore, a minus (−) value of the desired magnitude is passed.

The Reliant UNIX operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (*IPC_NOWAIT* flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process which is testing for a semaphore equal to zero; only read permission is required for this test; it is accomplished by passing a value of zero to the *semop* (semaphore operation) system call.

On the other hand, if the process is not successful and did 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 Reliant 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* system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is numbered one less than the total in the set.

A single system call can be used to perform a sequence of these "blocking/nonblocking operations" on a set of semaphores. When performing a sequence of operations, the blokking/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. For example, if the first three of six operations on a set of ten semaphores could be completed successfully, but the fourth operation would be blocked, no changes are made to the set until all six operations can be performed without blocking. Either the operations are successful and the semaphores are changed, or one ("nonblocking") operation is unsuccessful and none are changed. In short, the operations are "atomically performed".

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, an error code (–1) is returned to the process, and the external variable *errno* is set accordingly.

System calls (documented in [9]) make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable *errno* is set accordingly.

2.5.2.1 Using semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified data structure and semaphore set (array) must be created. The unique identifier is called the semaphore set identifier (*semid*); it is used to identify or refer to a particular data structure and semaphore set.

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

- semaphore value
- 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

There is one associated data structure for the uniquely identified semaphore set. This data structure contains the following information related to the semaphore set:

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

The C programming language data structure definition for the semaphore set (array member) is as follows:

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

It is located in the <*sys/sem.h*> header file. Likewise, the structure definition for the associated semaphore data

structure is as follows:

```
struct semid_ds
{
        struct ipc_perm sem_perm;   /* operation permission struct */
        struct sem      *sem_base;  /* ptr to first semaphore in set */
        ushort          sem_nsems;  /* # of semaphores in set */
time_t          sem_otime;  /* last semop time */
        long            sem_pad1;   /* reserved for time_t expansion */
time_t          sem_ctime;  /* last change time */
        long            sem_pad2;   /*time_t expansion */
        long            sem_pad3[4];/* reserve area */
};
```

It is also located in the <*sys/sem.h*> header file. Note that the *sem_perm* member of this structure uses *ipc_perm* as a template. The figure with ipc_perm data structure shows the breakout for the operation permissions data structure.

The *ipc_perm* data structure is the same for all IPC facilities; it is located in the <*sys/ipc.h*> header file and is shown in Section "Messages".

The *semget* system call is used to perform two tasks:

- to get a new semaphore set identifier and create an associated data structure and semaphore set for it
- to return an existing semaphore set identifier 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* at the security level of the calling process and the *IPC_CREAT* flag is set, a new *semid* is returned with an associated data structure and semaphore set created for it provided no system tunable parameter would be exceeded.

There is also a provision for specifying a *key* of value zero (0), which is known as the private *key* (*IPC_PRIVATE*). When specified, a new identifier at the security level of the creating process is always returned with an associated data structure and semaphore set created for it, unless a system-tunable parameter would be exceeded. The *ipcs* command will show the *key* field for the *semid* as all zeros.

When performing the first task, the process which calls *semget* becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator (see Section "Controlling semaphores"). The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a semaphore set identifier exists at the security level of the calling process for the *key* specified, the value of the existing identifier is returned. If you do not want to have an existing semaphore set identifier returned, a control command (*IPC_EXCL*) can be specified (set) in the *semflg* argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (nsems) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for *nsems*. (Using semget describes how to use this system call.)

Once a uniquely identified semaphore set and data structure are created, *semop* (semaphore operations) and *semctl* (semaphore control) can be used. Semaphore operations consist of incrementing, decrementing, and testing for zero. The *semop* system call is used to perform these operations (see Section "Operations on semaphores", for details of this system call).

The *semctl* system call permits you to control the semaphore facility in the following ways:

- by returning the value of a semaphore (*GETVAL*)
- by setting the value of a semaphore (*SETVAL*)
- by returning the PID of the last process performing an operation on a semaphore set (*GETPID*)
- by returning the number of processes waiting for a semaphore value to become greater than its current value (*GETNCNT*)

- by returning the number of processes waiting for a semaphore value to equal zero (*GETZCNT*)
- by getting all semaphore values in a set and placing them in an array in user memory (*GETALL*)
- by setting all semaphore values in a semaphore set from an array of values in user memory (*SETALL*)
- by retrieving the data structure associated with a semaphore set (*IPC_STAT*)
- by changing operation permissions for a semaphore set (*IPC_SET*)
- by removing a particular semaphore set identifier from the Reliant UNIX operating system along with its associated data structure and semaphore set (*IPC_RMID*)

See Section "Controlling semaphores", for details of the *semctl* system call.

2.5.2.2 Getting semaphores

This section describes how to use the *semget* system call. The accompanying program illustrates its use.

Using semget

The synopsis found in the *semget(2)* entry in [9] is as follows:

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

The following line in the synopsis:

```
int□semget□(key,□nsems,□semflg)
```

informs you that *semget* is a function with three formal arguments that returns an integer-type value. The next two lines:

```
key_t□□key;
int□nsems,□semflg;
```

declare the types of the formal arguments. *key_t* is defined by a *typedef* in the <*sys/types.h*> header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier that was discussed above.

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 created if either

- *key* is equal to *IPC_PRIVATE*, or
- *key* is a unique integer and *semflg* ANDed with *IPC_CREAT* is "true."

The value passed to the *semflg* argument must be an integer that will specify the following:

- operation permissions,
- control fields (commands).

The Table 18 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation permissions	Octal value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004

Alter by Others	00002

Table 18: Operation permissions codes

A specific value is derived by adding or bitwise ORing the values for the operation permissions wanted. 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 <*sys/sem.h*> header file which can be used for the user (OWNER). They are as follows:

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

Control flags are predefined constants (represented by all uppercase letters). The flags that apply to the *semget* system call are *IPC_CREAT* and *IPC_EXCL* and are defined in the <*sys/ipc.h*> header file.

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 adding or bitwise ORing (|) them with the operation permissions.

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

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

As specified by the *semget(2)* entry in [9], success or failure of this system call depends upon the actual argument values for *key*, *nsems*, and *semflg*, and system-tunable parameters.

The system call will attempt to return a new semaphore set identifier if one of the following conditions is true:

- *key* is equal to *IPC_PRIVATE*
- *key* does not already have a semaphore set identifier associated with it and (*semflg* & *IPC_CREAT*) is "true" (not zero).

The *key* argument can be set to *IPC_PRIVATE* like this:

semid = semget(IPC_PRIVATE, nsems, semflg);

SEMMNI, *SEMMNS*, or *SEMMSL* system-tunable parameters will always cause a failure. The *SEMMNI* system-tunable parameter determines the maximum number of unique semaphore sets (*semid*s) that may be in use at any given time. 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.

IPC_EXCL is another control command used in conjunction with *IPC_CREAT*. It will cause the system call to return an error if a semaphore set identifier already exists at the security level of the calling process for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) identifier when it has not. In other words, when both *IPC_CREAT* and *IPC_EXCL* are specified, a new semaphore set identifier is returned if the system call is successful. Any value for *semflg* returns a new identifier if the key equals zero (*IPC_PRIVATE*) and no system-tunable parameters are exceeded.

Refer to the *semget(2)* manual page in [9] for specific associated data structure initialization for successful completion. The specific failure conditions and their error names are contained there also.

Example program

The figure "semget system call example" is a menu-driven program. It allows all possible combinations of using the *semget* system call to be exercised.

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

This program begins (lines 4-8) by including the required header files as specified by the *semget(2)* entry in [9]. Note that the <*sys/errno.h*> header file is included as opposed to declaring *errno* as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self explanatory. These names make the program more readable and are perfectly legal since they are local to the program. The variables declared for this program and what they are used for are as follows:

key
: used to pass the value for the desired key

opperm
: used to store the desired operation permissions

flags
: used to store the desired control commands (flags)

opperm_flags
: used to store the combination from the logical ORing of the *opperm* and *flags* variables; it is then used in the system call to pass the *semflg* argument

semid
: used for returning the semaphore set identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions; the result is stored in *opperm_flags* (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57); its value is stored in *nsems*.

The system call is made next; the result is stored in the *semid* (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 (line 65). Remember that the external *errno* variable is only set when a system call fails; it should only be examined immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 69).

The example program for the *semget* system call follows. We suggest that you name the source program file *semget.c* and the executable file *semget*.

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

2.5.2.3 Controlling semaphores

This section describes how to use the *semctl* system call. The accompanying program illustrates its use.

Using semctl

The synopsis found in the *semctl(2)* entry in [9] is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
     int val;
     struct semid_ds *buf;
     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 sequences of operations (atomically performed) on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore is numbered one less than the total in the set.

The *cmd* argument can be replaced by one of the following values:

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 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 value 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 pointer.

IPC_SET
: for the specified semaphore set (*semid*), set the effective user/group identification and operation permissions.

IPC_RMID
: remove the specified semaphore set (*semid*) along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or system administrator to perform an *IPC_SET* or *IPC_RMID* control command. Read/alter permission is required as applicable for the other

control commands.

The *arg* argument is used to pass the system call the appropriate union member for the control command to be performed. For some of the control commands, the *arg* argument is not required and is simply ignored.

- *arg.val* required: *SETVAL*
- *arg.buf* required: *IPC_STAT*, *IPC_SET*
- *arg.array* required: *GETALL*, *SETALL*
- *arg* ignored: *GETVAL*, *GETPID*, *GETNCNT*, *GETZCNT*, *IPC_RMID*

The details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, read Using semget. It goes into more detail than would be practical to do for every system call.

Example program

The figure "semctl system call example" is a menu-driven program. It allows all possible combinations of using the *semctl* system call to be exercised.

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

This program begins (lines 5-9) by including the required header files as specified by the *semctl(2)* entry in [9]. Note that in this program errno is declared as an external variable, and therefore the <*sys/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 are perfectly legal since they are local to the program. Those declared for this program and what they are used for 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 119) 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 98-100) control command, and when initializing the arg.*array* when performing a *SETALL* (lines 117-121) control command

length
: used as a variable to test for the number of semaphores in a set against the *i* counter variable (lines 98, 117)

uid
: used to store the *IPC_SET* value for the user identification

gid
: used to store the *IPC_SET* value for the group identification

mode
: used to store the *IPC_SET* value for the operation permissions

retrn
: used to store the return value from the system call

semid
: used to store and pass the semaphore set identifier to the system call

semnum
: used to store and pass the semaphore number to the system call

cmd

used to store the code for the desired control command so that subsequent processing can be performed on it

choice
used to determine which member (*uid*, *gid*, *mode*) for the *IPC_SET* control command is to be changed

semvals[]
used to store the set of semaphore values when getting (*GETALL*) or initializing (*SETALL*)

arg.val
used to pass the system call a value to set, or to store a value returned from the system call, for a single semaphore (union member)

arg.buf
a pointer passed to the system call which locates the data structure in the user memory area where the *IPC_STAT* control command is to place its return values, or where the *IPC_SET* command gets the values to set (union member)

arg.array
a pointer passed to the system call which locates the array in the user memory where the *GETALL* control command is to place its return values, or when the *SETALL* command gets the values to set (union member)

Note that the *semid_ds* data structure in this program (line 14) uses the data structure located in the <*sys/sem.h*> header file of the same name as a template for its declaration.

Note that the *semvals* 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.

Now that all of the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored in the *semid* variable (lines 24-26). This is required for all *semctl* system calls.

Then, the code for the desired control command must be entered (lines 17-42), 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 48, 49). When it is entered, it is stored in the *semnum* variable (line 50). Then, the system call is performed, and the semaphore value is displayed (lines 51-54). Note that the *arg* argument is not required in this case, and the system call will simply ignore it.

If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 197, 198); if the system call is unsuccessful, an error message is displayed along with the value of the external *errno* variable (lines 194, 195).

If the *SETVAL* control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 55, 56). When it is entered, it is stored in the *semnum* variable (line 57). Next, a message prompts for the value to which the semaphore is to be set; it is stored as the *arg.val* member of the union (lines 58, 59). Then, the system call is performed (lines 60, 62). 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 63-66), and the PID of the process performing the last operation is displayed. Note that the *arg* argument is not required in this case, and the system call will simply ignore it. 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 67-71). 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 73-76). Note that the *arg* argument is not required in this case, and the system call will simply ignore it. 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 77-80). When it is entered, it is stored in the *semnum* variable (line 81). Then the system call is

performed and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 82-85). 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 87-93). The *length* variable is set to the number of semaphores in the set (line 93). The *arg.array* union member is set to point to the *semvals* array where the system call is to store the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the *arg.array* from zero to one less than the value of *length* (lines 98-104). 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 107-110). The *length* variable is set to the number of semaphores in the set (line 113). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the *semvals* array to contain the desired values of the semaphore set (lines 115-121). 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 *arg.array* union member is set to point to the *semvals* array from which the system call is to obtain the semaphore values. The system call is then made (lines 122-125). 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 129), and the status information returned is printed out (lines 130-141); 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 (line 194).

If the *IPC_SET* control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 145-149). 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 150-156). This code is stored in the *choice* variable (line 157). Now, depending upon the member picked, the program prompts for the new value (lines 158-181). The value is placed into the appropriate member in the user memory area data structure, and the system call is made (line 184). Depending upon success or failure, the program returns the same messages as for *GETVAL* above.

If the *IPC_RMID* control command (code 10) is selected, the system call is performed (lines 186-188). The semaphore set identifier along with its associated data structure and semaphore set is removed from the Reliant UNIX operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the *semctl* system call follows. We suggest that you name the source program file *semctl.c* and the executable file *semctl*.

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□□□□□□□□ushort□semvals[25];
□19□□□□□□□□union□semun□□{
□20□□□□□□□□□□□□□□int□val;
□21□□□□□□□□□□□□□□struct□semid_ds□*buf;
□22□□□□□□□□□□□□□□ushort□*array;
□23□□□□□□□□}□arg;
□24□□□□□□□□/*□Enter□the□semaphore□ID□*/
□25□□□□□□□□printf("Enter□the□semid□=□");
□26□□□□□□□□scanf("%d",□&semid);
□27□□□□□□□□/*□Choose□the□desired□command□*/
□28□□□□□□□□printf("\nEnter□the□number□for\n"
□29□□□□□□□□printf("the□desired□cmd:\n")
□30□□□□□□□□printf("GETVAL□□□□□□=□□1\n")
□31□□□□□□□□printf("SETVAL□□□□□□=□□2\n")
□32□□□□□□□□printf("GETPID□□□□□□=□□3\n")
□33□□□□□□□□printf("GETNCNT□□□□□=□□4\n")
□34□□□□□□□□printf("GETZCNT□□□□□=□□5\n")
□35□□□□□□□□printf("GETALL□□□□□□=□□6\n")
□36□□□□□□□□printf("SETALL□□□□□□=□□7\n")
□37□□□□□□□□printf("IPC_STAT□□□□=□□8\n")
□38□□□□□□□□printf("IPC_SET□□□□□=□□9\n")
□39□□□□□□□□printf("IPC_RMID□□□□=□□10\n")
□40□□□□□□□□printf("Entry□□□□□□□=□□");
□41□□□□□□□□scanf("%d",□&cmd);
□42□□□□□□□□/*□Check□entries□*/
□43□□□□□□□□printf□("\nsemid□=%d,□cmd□=□%d\n\
□44□□□□□□□□□□□□□□semid,□cmd);
□45□□□□□□□□/*□Set□the□command□and□do□the□call□*/
□46□□□□□□□□switch□(cmd)
□47□□□□□□□□{
□48□□□□□□□□case□1:□/*□Get□a□specified□value□*/
□49□□□□□□□□□□□□□□printf("\nEnter□the□semnum□=□")
□50□□□□□□□□□□□□□□scanf("%d",□&semnum);
□51□□□□□□□□□□□□□□/*□Aufruf□starten□*/
□52□□□□□□□□□□□□□□retrn□=□semctl(semid,□semnum,□GETVAL,□arg);
□53□□□□□□□□□□□□□□printf("\nThe□semval□=□%d",□retrn)
□54□□□□□□□□□□□□□□break;
□55□□□□□□□□case□2:□/*□Set□a□specified□value□*/
□56□□□□□□□□□□□□□□printf("\nEnter□the□semnum□=□")
□57□□□□□□□□□□□□□□scanf("%d",□&semnum);
□58□□□□□□□□□□□□□□printf("\nEnter□the□value□=□")
□59□□□□□□□□□□□□□□scanf("%d",□&arg.val);
□60□□□□□□□□□□□□□□/*□Aufruf□starten□*/
□61□□□□□□□□□□□□□□retrn□=□semctl(semid,□semnum,□SETVAL,□arg);
□62□□□□□□□□□□□□□□break;
□63□□□□□□□□case□3:□/*□Get□the□process□ID□*/
□64□□□□□□□□□□□□□□retrn□=□semctl(semid,□0,□GETPID,□arg);
□65□□□□□□□□□□□□□□printf("\nThe□sempid□=□%d",□retrn)
```

```
□66□□□□□□□□□□□□□break;
□67□□□□□□□□case□4:□/*□Get□the□number□of□processes
□68□□□□□□□□□□□□□□□□□□□□waiting□for□the□semaphore□to
□69□□□□□□□□□□□□□□□□□□□□become□greater□than□its□current
□70□□□□□□□□□□□□□□□□□□□□value□□□□□□□□□□□□□□□*/
□71□□□□□□□□□□□□□printf("\nEnter□the□semnum□=□")
□72□□□□□□□□□□□□□scanf("%d",□&semnum);
□73□□□□□□□□□□□□□/*□Do□the□system□call□*/
□74□□□□□□□□□□□□□retrn□=□semctl(semid,□semnum,□GETNCNT,□arg);
□75□□□□□□□□□□□□□printf("\nThe□semncnt□=□%d",□retrn)
□76□□□□□□□□□□□□□break;
□77□□□□□□□□case□5:□/*□Get□the□number□of□processes
□78□□□□□□□□□□□□□□□□□□□□waiting□for□the□semaphore
□79□□□□□□□□□□□□□□□□□□□□value□to□become□zero□□□□□□□□□□*/
□80□□□□□□□□□□□□□printf("\nEnter□the□semnum□=□")
□81□□□□□□□□□□□□□scanf("%d",□&semnum);
□82□□□□□□□□□□□□□/*□Do□the□system□call□*/
□83□□□□□□□□□□□□□retrn□=□semctl(semid,□semnum,□GETZCNT,□arg);
□84□□□□□□□□□□□□□printf("\nThe□semzcnt□=□%d",□retrn)
□85□□□□□□□□□□□□□break;
□86□□□□□□□□case□6:□/*□Get□all□of□the□semaphores□□*/
□87□□□□□□□□□□□□□□□□□□/*□Get□the□number□of□semaphores□in
□88□□□□□□□□□□□□□□□□□□□□the□semaphore□set□*/
□89□□□□□□□□□□□□□arg.buf□=□&semid_ds;
□90□□□□□□□□□□□□□retrn□=□semctl(semid,□0,□IPC_STAT,□arg);
□91□□□□□□□□□□□□□if(retrn□==□-1)
□92□□□□□□□□□□□□□□□□□□goto□ERROR;
□93□□□□□□□□□□□□□length□=□arg.buf->sem_nsems;
□94□□□□□□□□□□□□□/*□Get□and□print□all□semaphores□in□the
□95□□□□□□□□□□□□□□□□specified□set□*/
□96□□□□□□□□□□□□□arg.array□=□semvals;
□97□□□□□□□□□□□□□retrn□=□semctl(semid,□0,□GETALL,□arg);
□98□□□□□□□□□□□□□for□(i□=□0;□i□<□length;□i++)
□99□□□□□□□□□□□□□{
100□□□□□□□□□□□□□□□□printf("%d",□semvals[i]);
101□□□□□□□□□□□□□□□□/*□Separate□each
102□□□□□□□□□□□□□□□□□□□□semaphore□□*/
103□□□□□□□□□□□□□□□□printf("□");
104□□□□□□□□□□□□□}
105□□□□□□□□□□□□□break;
106□□□□□□□□case□7:□/*□Set□all□semaphores□in□the□set□□□*/
107□□□□□□□□□□□□□□□□/*□Get□the□number□of□semaphores□in
108□□□□□□□□□□□□□□□□□□□□the□set□*/
109□□□□□□□□□□□□□arg.buf□=□&semid_ds;
110□□□□□□□□□□□□□retrn□=□semctl(semid,□0,□IPC_STAT,□arg);
111□□□□□□□□□□□□□if(retrn□==□-1)
112□□□□□□□□□□□□□□□□□□goto□ERROR;
113□□□□□□□□□□□□□length□=□arg.buf->sem_nsems;
114□□□□□□□□□□□□□printf("Length□=□%d\n",□length)
115□□□□□□□□□□□□□/*□Set□the□semaphore□set□values□*/
116□□□□□□□□□□□□□printf("\nEnter□each□value:\n"
117□□□□□□□□□□□□□for(i□=□0;□i□<□length□;□i++)
118□□□□□□□□□□□□□{
119□□□□□□□□□□□□□□□□scanf("%d",□&c);
```

```
120                    semvals[i] = c;
121                }
122                /* Do the system call */
123                arg.array = semvals;
124                retrn = semctl(semid, 0, SETALL, arg);
125                break;
126            case 8: /* Get the status for the semaphore set */
127                    /* Get and print the current status values */
128                arg.buf = &semid_ds;
129                retrn = semctl(semid, 0, IPC_STAT, arg);
130                printf ("\nThe USER ID = %d\n
131                        arg.buf->sem_perm.uid);
132                printf ("The GROUP ID = %d\n"
133                        arg.buf->sem_perm.gid);
134                printf ("The operation permissions = 0%o\n"
135                        arg.buf->sem_perm.mode);
136                printf ("The number of semaphores in set = %d\n"
137                        arg.buf->sem_nsems);
138                printf ("The last semop time = %d\n"
139                        arg.buf->sem_otime);
140                printf ("The last change time  = %d\n"
141                        arg.buf->sem_ctime);
142                break;
143            case 9:    /* Select and change the desired
144                       member of the data structure */
145                /* Get the current status values */
146                arg.buf = &semid_ds;
147                retrn = semctl(semid, 0, IPC_STAT, arg.buf);
148                if(retrn == -1)
149                    goto ERROR;
150                /* Select the member to change */
151                printf("\nEnter the number for the\n"
152                printf("member to be changed:\n")
153                printf("sem_perm.uid   = 1\n")
154                printf("sem_perm.gid   = 2\n")
155                printf("sem_perm.mode  = 3\n")
156                printf("Entry          = ");
157                scanf("%d", &choice);
158                switch(choice){
159                case 1: /* Change the user ID */
160                    printf("\nEnter USER ID = ")
161                    scanf ("%d", &uid);
162                    arg.buf->sem_perm.uid = uid;
163                    printf("\nUSER ID = %d\n
164                            arg.buf->sem_perm.uid);
165                    break;
166                case 2: /* Change the group ID */
167                    printf("\nEnter GROUP ID = ")
168                    scanf("%d", &gid);
169                    arg.buf->sem_perm.gid = gid;
170                    printf("\nGROUP ID = %d\n
171                            arg.buf->sem_perm.gid);
172                    break;
173                case 3: /* Change the mode portion of
```

```
174□□□□□□□□□□□□□□□□□□□□□□□□□□□□the□operation
175□□□□□□□□□□□□□□□□□□□□□□□□□□□□permissions□□□□□□□□□□□□□□□□*/
176□□□□□□□□□□□□□□□□□printf("\nEnter□MODE□in□octal□=□")
177□□□□□□□□□□□□□□□□□scanf("%o",□&mode);
178□□□□□□□□□□□□□□□□□arg.buf->sem_perm.mode□=□mode;
179□□□□□□□□□□□□□□□□□printf("\nMODE□=□0%o\n
180□□□□□□□□□□□□□□□□□□□□□□□arg.buf->sem_perm.mode);
181□□□□□□□□□□□□□□□□□break;
182□□□□□□□□□□□□□}
183□□□□□□□□□□□□□/*□Do□the□change□*/
184□□□□□□□□□□□□□retrn□=□semctl(semid,□0,□IPC_SET,□arg);
185□□□□□□□□□□□□□break;
186□□□□□□□□case□10:□□□□/*□Remove□the□semid□along□with□its
187□□□□□□□□□□□□□□□□□□□□□□□□□data□structure□□*/
188□□□□□□□□□□□□□retrn□=□semctl(semid,□0,□IPC_RMID,□arg);
189□□□□□□□□}
190□□□□□□□□/*□Perform□the□following□if□the□call□is□unsuccessful:□*/
191□□□□□□□□if(retrn□==□-1)
192□□□□□□□□{
193□□□□ERROR:
194□□□□□□□□printf□("\nThe□semctl□call□failed!,□error□number□=□□%d\n",\□errno);
195□□□□□□□□□□□□□exit(0);
196□□□□□□□□}
197□□□□□□□□printf□("\n\nThe□semctl□system□call□was□successful\n
198□□□□□□□□printf□("for□semid□=□%d\n",□semid)
199□□□□□□□□exit□(0);
200□□□□}
```

2.5.2.4 Operations on semaphores

This section describes how to use the *semop* system call. The accompanying program illustrates its use.

Using semop

The synopsis found in the *semop(2)* entry in [9] is as follows:

```
#include□<sys/types.h>
#include□<sys/ipc.h>
#include□<sys/sem.h>
int□semop□(semid,□sops,□nsops)
int□semid;
struct□sembuf□*sops;
unsigned□nsops;
```

The *semop* system call requires three arguments to be passed to it and returns an integer value which will be zero for successful completion or -1 otherwise.

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 points to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number (*sem_num*)
- the operation to be performed (*sem_op*)
- the control flags (*sem_flg*)

The **sops* declaration means that either an array name (which is the address of the first element of the array) or a pointer to the array can be used. *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 <*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 (*sem_num*) 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:

- if *sem_op* is positive, the semaphore value is incremented by the value of *sem_op*
- if *sem_op* is negative, the semaphore value is decremented by the absolute value of *sem_op*
- if *sem_op* is zero, the semaphore value is tested for equality to zero

The following operation commands (flags) can be used:

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

SEM_UNDO—this operation command is used to tell the system to undo the process's semaphore changes automatically when the process exits; it allows processes to avoid deadlock problems. To implement this feature, the system maintains a table with an entry for every process in the system. Each entry points to a set of undo structures, one for each semaphore used by the process. The system records the net change.

Example program

The figure "semop system call example" is a menu-driven program. It allows all possible combinations of using the *semop* system call to be exercised.

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

This program begins (lines 5-9) by including the required header files as specified by the *shmop(2)* entry in [9]. Note that in this program *errno* is declared as an external variable; therefore, the <*sys/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 are perfectly legal since the declarations are local to the program. The variables declared for this program and what they are used for are as follows:

sembuf[10]
: used as an array buffer (line 14) to contain a maximum of ten *sembuf* type structures; ten is the standard value of the tunable parameter *SEMOPM*, the maximum number of operations on a semaphore set for each *semop* system call

sops
: used as a pointer (line 14) to the *sembuf* array for the system call and for accessing the structure members within the array

string[8]
: used as a character buffer to hold a number entered by the user

rtrn
: used to store the return value from the system call

flags
: used to store the code of the *IPC_NOWAIT* or *SEM_UNDO* flags for the *semop* system call (line 59)

sem_num
: used to store the semaphore number entered by the user for each semaphore operation in the array

i
: used as a counter (line 31) for initializing the structure members in the array, and used to print out each structure in the array (line 78)

semid
: used to store the desired semaphore set identifier for the system call

nsops
: used to specify the number of semaphore operations for the system call; must be less than or equal to *SEMOPM*

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 18-21). *semid* is stored in the *semid* variable (line 22).

A message is displayed requesting the number of operations to be performed on this set (lines 24-26). The number of operations is stored in the *nsops* variable (line 27).

Next, a loop is entered to initialize the array of structures (lines 29-76). 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 77-84).

The *sops* pointer is set to the address of the array (lines 85, 86). *sembuf* could be used directly, if desired, instead of *sops* in the system call.

The system call is made (line 88), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the *semctl GETALL* control command.

The example program for the *semop* system call follows. We suggest that you name the source program file *semop.c* and the executable file *semop*.

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[8];
□□□□16□□□□□□□□int□retrn,□flags,□sem_num,□i,□semid;
□□□□17□□□□□□□□unsigned□nsops;
□□□□18□□□□□□□□/*□Enter□the□semaphore□ID□*/
□□□□19□□□□□□□□printf("\nEnter□the□semid□of\n"
□□□□20□□□□□□□□printf("the□semaphore□set□to\n")
□□□□21□□□□□□□□printf("be□operated□on□=□");
□□□□22□□□□□□□□scanf("%d",□&semid);
□□□□23□□□□□□□□printf("\nsemid□=□%d",□semid)
□□□□24□□□□□□□□/*□Enter□the□number□of□operations□*/
□□□□25□□□□□□□□printf("\nEnter□the□number□of□semaphore\n"
□□□□26□□□□□□□□printf("operations□for□this□set□=□");
□□□□27□□□□□□□□scanf("%d",□&nsops);
□□□□28□□□□□□□□printf("\nsops□=□%d",□nsops)
□□□□29□□□□□□□□/*□Initialize□the□array□for□the
```

```
    30              number of operations to be performed */
    31          for(i = 0, sops = sembuf; i < nsops; i++, sops++)
    32          {
    33              /* This determines the semaphore in
    34                  the semaphore set      */
    35              printf("\nEnter the semaphore\n"
    36              printf("number (sem_num) = ");
    37              scanf("%d", &sem_num);
    38              sops->sem_num = sem_num;
    39              printf("\nThe sem_num = %d", sops->sem_num)
    40              /* Enter a (-)number to decrement,
    41                  an unsigned number (no +) to increment,
    42                  or zero to test for zero.  These values
    43                  are entered into a string and converted
    44                  to integer values.                    */
    45              printf("\nEnter the operation for\n"
    46              printf("the semaphore (sem_op) = ");
    47              scanf("%s", string);
    48              sops->sem_op = atoi(string);
    49              printf("\nsem_op = %d\n", sops->sem_op
    50              /* Specify the desired flags */
    51              printf("\nEnter the corresponding\n"
    52              printf("number for the desired\n")
    53              printf("flags:\n")
    54              printf("No flags                    = 0\n")
    55              printf("IPC_NOWAIT                  = 1\n")
    56              printf("SEM_UNDO                    = 2\n")
    57              printf("IPC_NOWAIT and SEM_UNDO     = 3\n")
    58              printf("            Flags           = ");
    59              scanf("%d", &flags);
    60              switch(flags)
    61              {
    62              case 0:
    63                  sops->sem_flg = 0;
    64                  break;
    65              case 1:
    66                  sops->sem_flg = IPC_NOWAIT;
    67                  break;
    68              case 2:
    69                  sops->sem_flg = SEM_UNDO;
    70                  break;
    71              case 3:
    72                  sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
    73                  break;
    74              }
    75              printf("\nFlags = 0%o\n", sops->sem_flg
    76          }
    77          /* Print out each structure in the array */
    78          for(i = 0; i < nsops; i++)
    79          {
    80              printf("\nsem_num = %d\n", sembuf[i].sem_num
    81              printf("sem_op = %d\n", sembuf[i].sem_op)
    82              printf("sem_flg = 0%o\n", sembuf[i].sem_flg)
    83              printf(" ");
```

```
□□□□84□□□□□□□□□}
□□□□85□□□□□□□□□sops□=□sembuf;□/*□Reset□the□pointer□to
□□□□86□□□□□□□□□□□□□□□□□□□□□□□□□□□□sembuf[0]□□□□□□□□□□□□*/
□□□□87□□□□□□□□□/*□Do□the□semop□system□call□*/
□□□□88□□□□□□□□□retrn□=□semop(semid,□sops,□nsops);
□□□□89□□□□□□□□□if(retrn□==□-1)□□{
□□□□90□□□□□□□□□□□□□□printf("\nSemop□failed,□error□=□%d\n",□errno
□□□□91□□□□□□□□□}
□□□□92□□□□□□□□□else□{
□□□□93□□□□□□□□□□□□□□printf□("\nSemop□was□successful\n"
□□□□94□□□□□□□□□□□□□□printf("for□semid□=□%d\n",□semid)
□□□□95□□□□□□□□□□□□□□printf("Value□returned□=□%d\n",□retrn)
□□□□96□□□□□□□□□}
□□□□97□□□□}
```

2.5.3 Shared memory

The shared memory type of IPC allows two or more processes (executing programs) to share memory and, consequently, the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware-dependent.

This sharing of memory provides the fastest means of exchanging data between processes. However, processes that reference a shared memory segment must reside on one processor. Processes running on different processors (such as in an NFS network) cannot use shared memory segments.

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.

shmat (shared memory attach) and *shmdt* (shared memory detach) can be performed on a shared memory segment.

shmat allows processes to associate themselves with the shared memory segment if they have permission. segment if they have permission and are at the security level of the segment. They can then read or write as allowed.

shmdt 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 [9]) make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (−1) is returned to the process, and the external variable *errno* is set accordingly.

2.5.3.1 Using shared memory

Sharing memory between processes occurs on a virtual segment basis. There is only one copy of each individual shared memory segment existing in the Reliant UNIX operating system at any point in time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (*shmid*); it is used to identify or refer to the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size
- segment descriptor (for internal system use only)
- PID performing last operation
- PID of creator
- current number of processes attached
- last attach time
- last detach time
- last change time

The C programming language data structure definition for the shared memory segment data structure is located in the <*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□*/
□□□□□pid_t□□□□□□□□□□□□□□shm_lpid;□□□□□□/*□pid□of□last□shmop□*/
□□□□□pid_t□□□□□□□□□□□□□□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; is it located in the <*sys/ipc.h*> header file and shown in the figure with ipc_perm data structure.

The *shmget* system call performs two tasks:

- it gets a new shared memory identifier and creates an associated shared memory segment data structure for it
- it returns an existing shared memory identifier 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 shared memory identifier and the *IPC_CREAT* flag is set in *shmflg*, a new identifier is returned with an associated shared memory segment data structure created for it provided no system-tunable parameters would be exceeded.

There is also a provision for specifying a *key* of value zero which is known as the private *key* (*IPC_PRIVATE*); 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. The *ipcs* command will show the *key* field for the *shmid* as all zeros.

For the second task, if a *shmid* exists for the key specified, the value of the existing *shmid* is returned. If it is not desired to have an existing *shmid* returned, a control command (*IPC_EXCL*) can be specified (set) in the *shmflg* argument passed to the system call. Using shmget discusses how to use this system call.

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 Section "Controlling shared memory"). 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, *shmop* (shared memory segment operations) and *shmctl* (shared memory control) can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. *shmat* and *shmdt* are provided for each of these operations (see Section "Operations for shared memory", for details on these system calls).

The *shmctl* system call permits you to control the shared memory facility in the following ways:

- by retrieving the data structure associated with a shared memory segment (*IPC_STAT*)
- by changing operation permissions for a shared memory segment (*IPC_SET*)
- by removing a particular shared memory segment from the Reliant UNIX operating system along with its associated shared memory segment data structure (*IPC_RMID*)
- by locking a shared memory segment in memory (*SHM_LOCK*)
- by unlocking a shared memory segment (*SHM_UNLOCK*)

See Section "Controlling shared memory", for details of the *shmctl* system call.

2.5.3.2 Getting shared memory segments

This section describes how to use the *shmget* system call. The accompanying program illustrates its use.

Using shmget

The synopsis found in the *shmget(2)* entry in [9] 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 Reliant UNIX operating system. The following line in the synopsis:

```
int□shmget□(key,□size,□shmflg)
```

informs you that *shmget* is a function with three formal arguments that returns an integer-type value. The next two lines:

```
key_t□□key;
int□size,□shmflg;
```

declare the types of the formal arguments. *key_t* is defined by a typedef in the <*sys/types.h*> header file to be an integer.

The integer returned from this function (upon successful completion) is the shared memory identifier (*shmid*) that was discussed earlier.

As declared, the process calling the *shmget* system call must supply three arguments to be passed to the formal *key*, *size*, and *shmflg* arguments.

A new *shmid* with an associated shared memory data structure is provided if either

- *key* is equal to *IPC_PRIVATE*, or
- *key* is a unique integer and *shmflg* ANDed with *IPC_CREAT* is "true" (not zero).

The value passed to the *shmflg* argument must be an integer-type value and will specify the following:

- operations permissions
- control fields (commands)

Access permissions determine the read/write attributes and modes determine the user/group/other attributes of the *shmflg* argument. They are collectively referred to as "Operation permissions" on .... The Table 19

reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation permissions	Octal value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Table 19: Operation permissions codes

A specific octal value is derived by adding or bitwise ORing the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the <*sys/shm.h*> header file which can be used for the user (OWNER). They are:

SHM_R 0400
SHM_W 0200

Control flags are predefined constants (represented by all uppercase letters). The flags that apply to the *shmget* system call are *IPC_CREAT* and *IPC_EXCL* and are defined in the

<*sys/ipc.h*> header file.

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 adding or bitwise ORing (|) them with the operation permissions. The *shmflg* value can easily be set by using the names of the flags in conjunction with the octal operation permissions value:

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

As specified by the *shmget(2)* entry in [9], success or failure of this system call depends upon the argument values for *key*, *size*, and *shmflg*, and system-tunable parameters. The system call will attempt to return a new *shmid* if one of the following conditions is true:

- *key* is equal to *IPC_PRIVATE* .
- *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* like this:

shmid = shmget(IPC_PRIVATE, size, shmflg);

The *SHMMNI* system-tunable parameter determines the maximum number of unique shared memory segments (*shmid*s) that may be in use at any given time. If the maximum number of shared memory segments is already in use, an attempt to create an additional segment will fail.

IPC_EXCL is another control command used in conjunction with *IPC_CREAT*. It will cause the system call to retrieve an error if a shared memory identifier 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 *PC_CREAT* and *IPC_EXCL* are specified, a unique shared memory identifier is returned if the system call is successful. Any value for *shmflg* returns a new identifier if the *key* equals zero (*IPC_PRIVATE*) and no system-tunable parameters are exceeded.

The system call will fail if the value for the *size* argument is less than *SHMMIN* or greater than *SHMMAX*. These tunable parameters specify the minimum and maximum shared memory segment sizes.

Refer to the *shmget(2)* manual page in [9] for specific associated data structure initialization for successful completion. The specific failure conditions and their error names are contained there also.

Example program

The figure "shmget system call example" is a menu-driven program. It allows all possible combinations of using the *shmget* system call to be exercised.

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

This program begins (lines 4-7) by including the required header files as specified by the *shmget(2)* entry in [9]. Note that the <*sys/errno.h*> header file is included as opposed to declaring *errno* as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self explanatory. These names make the program more readable and are perfectly legal since they are local to the program. The variables declared for this program and what they are used for 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)

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

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

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions; the result is stored in the *opperm_flags* variable (lines 35-50). A display then prompts for the size of the shared memory segment; it is stored in the *size* variable (lines 51-54).

The system call is made next; 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 (line 60). If no error occurred, the returned shared memory segment identifier is displayed (line 64).

The example program for the *shmget* system call follows. We suggest that you name the source program file *shmget.c* and the executable file *shmget*.

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);
□□□□33□□□□□□□□□/*□Check□the□values□*/
□□□□33□□□□□□□□□printf□("\nkey□=0x%x,□opperm□=□0%o,□flags□=□%d\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□call□failed,□error□number□=□%d\n",\□errno);
□□□□61□□□□□□□□}
□□□□62□□□□□□□□/*□Return□the□shmid□upon□successful□completion□*/
```

```
□□□□63□□□□□□□□else
□□□□64□□□□□□□□□printf□("\nThe□shmid□=□%d\n",□shmid
□□□□65□□□□□□□□exit(0);
□□□□66□□□□}
```

2.5.3.3 Controlling shared memory

This section describes how to use the *shmctl* system call. The accompanying program illustrates its use.

Using shmctl

The synopsis found in the *shmctl(2)* entry in [9] 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. It returns an integer value which will be zero for successful completion or -1 otherwise.

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 values:

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* with its associated shared memory segment data structure

SHM_LOCK
: lock the specified shared memory segment in memory; must be the system administrator to perform this operation

SHM_LOCK
: lock the shared memory segment from memory; must be system administrator to perform this operation

A process must have an effective user identification of OWNER/CREATOR or system administrator to perform an *IPC_SET* or *IPC_RMID* control command. Only the system administrator 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. If you need more information on the logic manipulations in this program, read Using shmget". It goes into more detail than what would be practical for every system call.

Example program

The figure "shmctl system call example" is a menu-driven program. It allows all possible combinations of using the *shmctl* system call to be exercised.

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

This program begins (lines 5-9) by including the required header files as specified by the *shmctl(2)* entry in [9]. Note that in this program *errno* is declared as an external variable, and therefore, the <*sys/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 are

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

uid
: used to store the *IPC_SET* value for the user identification

gid
: used to store the *IPC_SET* value for the 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 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 of the same name located in the <*sys/shm.h*> header file as a template for its declaration.

The next important thing to observe is that although the *buf* pointer is declared to be a pointer to a data structure of the *shmid_ds* type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier which is stored 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); 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 139), 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 141). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 143-147).

If the *IPC_SET* control command is selected (code 2), the first thing done is to get the current status information for the shared memory identifier specified (lines 88-90). 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 91-96). This code is stored in the *choice* variable (line 97). Now, depending upon the member picked, the program prompts for the new value (lines 98-120). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (lines 121-128). 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 125-128), and the *shmid* along with its associated message queue and data structure are removed from the Reliant UNIX operating system. Note that the *buf* pointer is ignored in performing 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 130,131). 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 133-135). 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 follows. We suggest that you name the source program file *shmctl.c* and the executable file *shmctl*.

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□get
□□□□□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□sequence\n");
□□□□□52□□□□□□□□□□□□□□printf□("number□=□0%x\n",
□□□□□53□□□□□□□□□□□□□□□□□□buf–>shm_perm.seq);
□□□□□54□□□□□□□□□□□□□□printf□("The□key=□0%x\n",
□□□□□55□□□□□□□□□□□□□□□□□□buf–>shm_perm.key);
□□□□□56□□□□□□□□□□□□□□printf□("The□segment□size□=□%d\n",
□□□□□57□□□□□□□□□□□□□□□□□□buf–>shm_segsz);
□□□□□58□□□□□□□□□□□□□□printf□("The□pid□of□last□shmop□=□%d\n",
□□□□□59□□□□□□□□□□□□□□□□□□buf–>shm_lpid);
□□□□□60□□□□□□□□□□□□□□printf□("The□pid□of□creator□=□%d\n",
□□□□□61□□□□□□□□□□□□□□□□□□buf–>shm_cpid);
□□□□□62□□□□□□□□□□□□□□printf□("The□current□#□attached□=□%d\n",
□□□□□63□□□□□□□□□□□□□□□□□□buf–>shm_nattch);
□□□□□64□□□□□□□□□□□□□□printf("The□last□shmat□time□=□%ld\n",
□□□□□65□□□□□□□□□□□□□□□□□□buf->shm_atime);
□□□□□66□□□□□□□□□□□□□□printf("The□last□shmdt□time□=□%ld\n",
□□□□□67□□□□□□□□□□□□□□□□□□buf->shm_dtime);
□□□□□68□□□□□□□□□□□□□□printf("The□last□change□time□=□%ld\n",
□□□□□69□□□□□□□□□□□□□□□□□□buf->shm_ctime);
□□□□□70□□□□□□□□□□□□□□break;
□□□□□□□□□□□□□□□□□□□□/*□Lines□71□-□85□deleted□*/
□□□□□86□□□□□□□□□case□2:□□□□/*□Select□and□change□the□desired
□□□□□87□□□□□□□□□□□□□□□□□□□□□□□□member(s)□of□the□data□structure.□*/
□□□□□88□□□□□□□□□□□□□□/*□Get□the□original□data□for□this□shmid
□□□□□89□□□□□□□□□□□□□□□□□□□□data□structure□first.□*/
□□□□□90□□□□□□□□□□□□□□rtrn□=□shmctl(shmid,□IPC_STAT,□buf);
□□□□□91□□□□□□□□□□□□□□printf("\nEnter□the□number□for□the\n");
□□□□□92□□□□□□□□□□□□□□printf("member□to□be□changed:\n");
□□□□□93□□□□□□□□□□□□□□printf("shm_perm.uid□□□=□1\n");
□□□□□94□□□□□□□□□□□□□□printf("shm_perm.gid□□□=□2\n");
□□□□□95□□□□□□□□□□□□□□printf("shm_perm.mode□□=□3\n");
□□□□□96□□□□□□□□□□□□□□printf("Entry□□□□□□□□□□=□");
□□□□□97□□□□□□□□□□□□□□scanf("%d",□&choice);
□□□□□98□□□□□□□□□□□□□□switch(choice){
□□□□□99□□□□□□□□□□□□□□case□1:
□□□□100□□□□□□□□□□□□□□□□printf("\nEnter□USER□ID□=□");
□□□□101□□□□□□□□□□□□□□□□scanf□("%d",□&uid);
□□□□102□□□□□□□□□□□□□□□□buf–>shm_perm.uid□=□uid;
□□□□103□□□□□□□□□□□□□□□□printf("\nUSER□ID□=□%d\n",
□□□□104□□□□□□□□□□□□□□□□□□□□buf–>shm_perm.uid);
□□□□105□□□□□□□□□□□□□□□□break;
□□□□106□□□□□□□□□□□□□□□case□2:
```

```
□□□□107□□□□□□□□□□□□□□□□□printf("\nEnter□GROUP□ID□=□");
□□□□108□□□□□□□□□□□□□□□□□scanf("%d",□&gid);
□□□□109□□□□□□□□□□□□□□□□□buf–>shm_perm.gid□=□gid;
□□□□110□□□□□□□□□□□□□□□□□printf("\nGROUP□ID□=□%d\n",
□□□□111□□□□□□□□□□□□□□□□□□□□□buf–>shm_perm.gid);
□□□□112□□□□□□□□□□□□□□□□□break;
□□□□113□□□□□□□□□□□□□case□3:
□□□□114□□□□□□□□□□□□□□□□□printf("\nEnter□MODE□in□octal□=□");
□□□□115□□□□□□□□□□□□□□□□□scanf("%o",□&mode);
□□□□116□□□□□□□□□□□□□□□□□buf–>shm_perm.mode□=□mode;
□□□□117□□□□□□□□□□□□□□□□□printf("\nMODE□=□0%o\n",
□□□□118□□□□□□□□□□□□□□□□□□□□□buf–>shm_perm.mode);
□□□□119□□□□□□□□□□□□□□□□□break;
□□□□120□□□□□□□□□□□□□}
□□□□121□□□□□□□□□□□□□/*□Do□the□change.□*/
□□□□122□□□□□□□□□□□□□rtrn□=□shmctl(shmid,□IPC_SET,
□□□□123□□□□□□□□□□□□□□□□□buf);
□□□□124□□□□□□□□□□□□□break;
□□□□125□□□□□□□□□case□3:□□□□/*□Remove□the□shmid□along□with□its
□□□□126□□□□□□□□□□□□□□□□□□□□□□associated
□□□□127□□□□□□□□□□□□□□□□□□□□□□data□structure.□*/
□□□□128□□□□□□□□□□□□□rtrn□=□shmctl(shmid,□IPC_RMID,□(struct□shmid_ds□*)\□NULL);
□□□□129□□□□□□□□□□□□□break;
□□□□130□□□□□□□□□case□4:□/*□Lock□the□shared□memory□segment□*/
□□□□131□□□□□□□□□□□□□rtrn□=□shmctl(shmid,□SHM_LOCK,□(struct□shmid_ds□*)\□NULL);
□□□□132□□□□□□□□□□□□□break;
□□□□133□□□□□□□□□case□5:□/*□Unlock□the□shared□memory
□□□□134□□□□□□□□□□□□□□□□□□□□□□segment.□*/
□□□□135□□□□□□□□□□□□□rtrn□=□shmctl(shmid,□SHM_UNLOCK,(struct□shmid_ds□*)\□NULL);
□□□□136□□□□□□□□□□□□□break;
□□□□137□□□□□□□□□}
□□□138□□□□□□/*□Perform□the□following□if□the□call□is□unsuccessful:□*/
□□□139□□□□□□if(rtrn□==□-1)
□□□140□□□□□□{
□□□141□□□□□□□□printf□("\nThe□shmctl□call□failed,□error□number□=□%d\n",\□errno);
□□□142□□□□□□}
□□□143□□□□□□/*□Return□the□shmid□upon□successful□completion□*/
□□□144□□□□□□else
□□□145□□□□□□□□printf□("\nShmctl□was□successful□for□shmid□=□%d\n
□□□146□□□□□□□□□□□□□□shmid);
□□□147□□□□□□exit□(0);
□□□148□□□□□}
□□□138□□□□□□/*□Perform□the□following□if□the□call□is□unsuccessful:□*/
□□□139□□□□□□if(rtrn□==□-1)
□□□140□□□□□□{
□□□141□□□□□□□□printf□("\nThe□shmctl□call□failed,□error□number□=□%d\n",\□errno);
□□□142□□□□□□}
□□□143□□□□□□/*□Return□the□shmid□upon□successful□completion□*/
□□□144□□□□□□else
□□□145□□□□□□□□printf□("\nShmctl□was□successful□for□shmid□=□%d\n
□□□146□□□□□□□□□□□□□□shmid);
□□□147□□□□□□exit□(0);
□□□148□□□□□}
```

2.5.3.4 Operations for shared memory

This section describes how to use the *shmat* and *shmdt* system calls.

Using shmop

The synopsis found in the *shmop(2)* entry in [9] is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;
int shmdt (shmaddr)
char *shmaddr;
```

Attaching a shared memory segment

The *shmat* system call requires three arguments to be passed to it. It returns a character pointer value. Upon successful completion, this value will be the address in memory where the process is attached to the shared memory segment and when unsuccessful the value will be −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 Reliant UNIX operating system picks the address where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the Reliant UNIX operating system would pick. The following illustrates some typical address ranges.

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

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

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

Detaching shared memory segments

The *shmdt* system call requires one argument to be passed to it. It returns an integer value which will be zero for successful completion or -1 otherwise.

Further details on *shmat* and *shmdt* are discussed in the example program. If you need more information on the logic manipulations in this program, read Using shmget. It goes into more detail than would be practical to do for every system call.

Example program

The figure "shmop system call example" is a menu-driven program. It allows all possible combinations of using the *shmat* and *shmdt* system calls to be exercised.

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

This program begins (lines 5-9) by including the required header files as specified by the *shmop(2)* entry in [9]. Note that in this program *errno* is declared as an external variable; therefore, the <*sys/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 are perfectly legal since they are local to the program. The variables declared for this program and what they are used for are as follows:

addr
: used to store the address of the shared memory segment for the *shmat* and *shmdt* system calls and to receive the return value from the *shmat* system call

laddr
: used to store the desired attach/detach address entered by the user

flags
: used to store the codes of the *SHM_RND* or *SHM_RDONLY* flags for the *shmat* system call 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

retrn
: used to store the return values from the *shmdt* system call

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 at the address of the *attach* variable (lines 19-23).

A loop is entered using the *attach* variable and the *i* counter (lines 23-72) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 26-29); it is stored in the *shmid* variable (line 30). Next, the program prompts for the address where the segment is to be attached (lines 32-36); it is stored in the *laddr* variable (line 37) and converted to a pointer (line 39). Then, the program prompts for the desired flags to be used for the attachment (lines 40-47), and the code representing the flags is stored in the *flags* variable (line 48). 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 49-60). The system call is executed (line 63). If successful, a message stating so is displayed along with the attach address (lines 68-70). If unsuccessful, a message stating so is displayed and the error code is displayed (line 65). 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 73-77) and the value is stored in the *detach* variable (line 76).

A loop is entered using the *detach* variable and the *i* counter (lines 80-98) 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 81-85); it is stored in the *laddr* variable (line 86) and converted to a pointer (line 88). Then, the *shmdt* system call is performed (line 89). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 95, 96). If unsuccessful, the error number is displayed (line 92). The loop continues until it finishes.

The example program for the *shmop* system calls follows. We suggest that you name the source program file *shmop.c* and the executable file *shmop*.

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□□□□□□□□char□*addr;
□□□□□15□□□□□□□□long□laddr;
□□□□□16□□□□□□□□int□flags,□i,□attach;
□□□□□17□□□□□□□□int□shmid,□shmflg,□retrn,□detach;
□□□□□18□□□□□□□□/*□Loop□for□attachments□by□this□process.□*/
□□□□□19□□□□□□□□printf("Enter□the□number□of\n");
□□□□□20□□□□□□□□printf("attachments□for□this\n");
□□□□□21□□□□□□□□printf("process□(1-4).\n");
□□□□□22□□□□□□□□printf("□□□□□□□□Attachments□=□");
□□□□□23□□□□□□□□scanf("%d",□&attach);
□□□□□24□□□□□□□□printf("Number□of□attaches□=□%d\n",□attach);
□□□□□25□□□□□□□□for(i□=□1;□i□<=□attach;□i++)□{
□□□□□26□□□□□□□□□□□□□□/*□Enter□the□shared□memory□ID.□*/
□□□□□27□□□□□□□□□□□□□□printf("\nEnter□the□shmid□of\n");
□□□□□28□□□□□□□□□□□□□□printf("the□shared□memory□segment□to\n");
□□□□□29□□□□□□□□□□□□□□printf("be□operated□on□=□");
□□□□□30□□□□□□□□□□□□□□scanf("%d",□&shmid);
□□□□□31□□□□□□□□□□□□□□printf("\nshmid□=□%d\n",□shmid);
□□□□□32□□□□□□□□□□□□□□/*□Enter□the□value□for□shmaddr.□*/
□□□□□33□□□□□□□□□□□□□□printf("\nEnter□the□value□for\n");
□□□□□34□□□□□□□□□□□□□□printf("the□shared□memory□address\n");
□□□□□35□□□□□□□□□□□□□□printf("in□hexadecimal:\n");
□□□□□36□□□□□□□□□□□□□□printf("□□□□□□□□□□□□□□Shmaddr□=□");
□□□□□37□□□□□□□□□□□□□□scanf("%lx",□&laddr);
□□□□□38□□□□□□□□□□□□□□addr□=□(char*)□laddr;
□□□□□39□□□□□□□□□□□□□□printf("The□desired□address□=□0x%lx\n",□(long)addr);
□□□□□40□□□□□□□□□□□□□□/*□Specify□the□desired□flags.□*/
□□□□□41□□□□□□□□□□□□□□printf("\nEnter□the□corresponding\n");
□□□□□42□□□□□□□□□□□□□□printf("number□for□the□desired\n");
□□□□□43□□□□□□□□□□□□□□printf("flags:\n");
□□□□□44□□□□□□□□□□□□□□printf("SHM_RND□□□□□□□□□□□□□□□□□□□□=□1\n");
□□□□□45□□□□□□□□□□□□□□printf("SHM_RDONLY□□□□□□□□□□□□□□□=□2\n");
□□□□□46□□□□□□□□□□□□□□printf("SHM_RND□and□SHM_RDONLY□=□3\n");
□□□□□47□□□□□□□□□□□□□□printf("□□□□□□□□□□□□□□Flags□□□□□□□=□");
□□□□□48□□□□□□□□□□□□□□scanf("%d",□&flags);
□□□□□49□□□□□□□□□□□□□□switch(flags)
□□□□□50□□□□□□□□□□□□□□{
□□□□□51□□□□□□□□□□□□□□case□1:
□□□□□52□□□□□□□□□□□□□□□□□□shmflg□=□SHM_RND;
```

```
□□□□□53□□□□□□□□□□□□□□□□□break;
□□□□□54□□□□□□□□□□□□□case□2:
□□□□□55□□□□□□□□□□□□□□□□shmflg□=□SHM_RDONLY;
□□□□□56□□□□□□□□□□□□□□□□break;
□□□□□57□□□□□□□□□□□□□case□3:
□□□□□58□□□□□□□□□□□□□□□□shmflg□=□SHM_RND□|□SHM_RDONLY;
□□□□□59□□□□□□□□□□□□□□□□break;
□□□□□60□□□□□□□□□□□□□}
□□□□□61□□□□□□□□□□□□□printf("\nFlags□=□0%o\n",□shmflg);
□□□□□62□□□□□□□□□□□□□/*□Do□the□shmat□system□call.□*/
□□□□□63□□□□□□□□□□□□□addr□=□shmat(shmid,□addr,□shmflg);
□□□□□64□□□□□□□□□□□□□if(addr□==□(char*)□-1)□{
□□□□□65□□□□□□□□□□□□□□□□printf("\nShmat□failed,□error□=□%d\n",□errno);
□□□□□66□□□□□□□□□□□□□}
□□□□□67□□□□□□□□□□□□□else□{
□□□□□68□□□□□□□□□□□□□□□□printf□("\nShmat□was□successful\n");
□□□□□69□□□□□□□□□□□□□□□□printf("for□shmid□=□%d\n",□shmid);
□□□□□70□□□□□□□□□□□□□□□□printf("The□address□=□0x%lx\n",□(long)addr);
□□□□□71□□□□□□□□□□□□□}
□□□□□72□□□□□□□□□}
□□□□□73□□□□□□□□□/*□Loop□for□detachments□by□this□process.□*/
□□□□□74□□□□□□□□□printf("Enter□the□number□of\n");
□□□□□75□□□□□□□□□printf("detachments□for□this\n");
□□□□□76□□□□□□□□□printf("process□(1-4).\n");
□□□□□77□□□□□□□□□printf("□□□□□□□□Detachments□=□");
□□□□□78□□□□□□□□□scanf("%d",□&detach);
□□□□□79□□□□□□□□□printf("Number□of□attaches□=□%d\n",□detach);
□□□□□80□□□□□□□□□for(i□=□1;□i□<=□detach;□i++)□{
□□□□□82□□□□□□□□□□□□□printf("\nEnter□the□value□for\n");
□□□□□83□□□□□□□□□□□□□printf("the□shared□memory□address\n");
□□□□□84□□□□□□□□□□□□□printf("in□hexadecimal:\n");
□□□□□85□□□□□□□□□□□□□printf("□□□□□□□□□□□□□Shmaddr□=□");
□□□□□86□□□□□□□□□□□□□scanf("%lx",□&laddr);
□□□□□87□□□□□□□□□□□□□addr□=□(char*)laddr;
□□□□□88□□□□□□□□□□□□□printf("The□desired□address□=□0x%lx\n",□(long)addr);
□□□□□89□□□□□□□□□□□□□/*□Do□the□shmdt□system□call.□*/
□□□□□90□□□□□□□□□□□□□retrn□=□shmdt(addr);
□□□□□91□□□□□□□□□□□□□if(retrn□==□-1)□□{
□□□□□92□□□□□□□□□□□□□□□□printf("Error□=□%d\n",□errno);
□□□□□93□□□□□□□□□□□□□}
□□□□□94□□□□□□□□□□□□□else□{
□□□□□95□□□□□□□□□□□□□□□□printf□("\nShmdt□was□successful\n");
□□□□□96□□□□□□□□□□□□□□□□printf("for□address□□=□0x%lx\n",□(long)addr);
□□□□□97□□□□□□□□□□□□□}
□□□□□98□□□□□□□□□}
□□□□□99□□□□□}
```

3 External interfaces and links

3.1 Memory management

The Reliant UNIX system provides a complete set of memory management mechanisms, providing applications complete control over the construction of their address space and permitting a wide variety of operations on both process address spaces and the variety of memory objects in the system. Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents the user with mappings that simulate the traditional Reliant UNIX process memory environment, but other views of memory are useful as well.

The Reliant UNIX memory-management facilities:

- Unify the system's operations on memory.
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support.
- Maintain consistency with the existing environment, in particular using the Reliant UNIX file system as the name space for named virtual-memory objects.

3.1.1 Virtual memory, address spaces and mapping

The system's "virtual memory" (VM) consists of all available physical memory resources. Examples include local and remote file systems, processor primary memory, swap space, and other random-access devices. Named objects in the virtual memory are referenced though the Reliant UNIX file system. However, not all file system objects are in the virtual memory; devices that cannot be treated as storage, such as terminal and network device files, are not in the virtual memory. Some virtual memory objects, such as private process memory and shared memory segments, do not have names.

A process's "address space" is defined by mappings onto objects in the system's virtual memory (usually files). Each mapping is constrained to be sized and aligned with the page boundaries of the system on which the process is executing. Each page may be mapped (or not) independently. Only process addresses which are mapped to some system object are valid, for there is no memory associated with processes themselves—all memory is represented by objects in the system's virtual memory.

Each object in the virtual memory has an "object address space" defined by some physical storage. A reference to an object address accesses the physical storage to which the object address is mapped. The virtual memory's associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made is not affected by the mere "existence" of the mapping.

Thus, it cannot, in general, be expected that an object has an "awareness" of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a 'page' is not a property of the object. Establishing a mapping to an object simply provides the "potential" for a process to access or change the object's contents.

The establishment of mappings provides an "access method" that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through *read* and *write*. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the *read*, modify buffer, *write* cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

3.1.2 Networking, heterogeneity and coherence

VM is designed to fit well with the larger Reliant UNIX heterogeneous environment. This environment makes extensive use of networking to access file systems—file systems that are now part of the system's virtual memory. Networks are not constrained to consist of similar hardware or to be based upon a common

operating system; in fact, the opposite is encouraged, for such constraints create serious barriers to accommodating heterogeneity. While a given set of processes may "apply" a set of mechanisms to establish and maintain the properties of various system objects—properties such as page sizes and the ability of objects to synchronize their own use—a given operating system should not "impose" such mechanisms on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running the Reliant UNIX memory management system and also to be accessed by systems for which virtual memory and storage management techniques such as paging are totally foreign, such as PC-DOS. Such systems can continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The unacceptable alternative would be to prohibit access to the object by less capable systems.

Another consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In both of these cases, the object is being shared, and thus the applications must use some synchronization mechanism to guarantee the coherence of their transactions with it. The scope and nature of the synchronization mechanism is best left to the application to decide. For example, file access on systems which do not support virtual memory access methods must be indirect, by way of *read* and *write*. Applications sharing files on such systems must coordinate their access using semaphores, file locking, or some application-specific protocols. What is required in an environment where mapping replaces *read* and *write* as the access method is an operation, such as *fsync*, that supports atomic update operations.

The nature and scope of synchronization over shared objects is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing. By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, the needs of applications are met without erecting barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the operating system.

3.1.3 Memory management interfaces

The applications programmer gains access to the facilities of the VM system through several sets of system calls. This section summarizes these calls, and provides examples of their use. For details, see [9].

3.1.3.1 Creating and using mappings

```
caddr_t
mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t off);
```

mmap establishes a mapping between a process's address space and an object in the system's virtual memory. It is the system's most fundamental function for defining the contents of an address space — all other system functions that contribute to the definition of an address space are built from *mmap*. The format of an *mmap* call is:

```
paddr = mmap(addr, len, prot, flags, fd, off);
```

mmap establishes a mapping from the process's address space at an address *paddr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. The value returned by *mmap* is an implementation-dependent function of the parameter *addr* and the setting of the *MAP_FIXED* bit of *flags*,as described below. A successful call to *mmap* returns *paddr* as its result. The address range [*paddr*, *paddr* + *len*) must be valid for the address space of the process and the range [*off*, *off* + *len*) must be valid for the virtual memory object. (The notation [*start*, *end*) refers to the interval from *start* to *end*,including *start* but not including *end*.)

The mapping established by *mmap* replaces any previous mappings for the process's pages in the range [*paddr*, *paddr* + *len*).

The parameter *prot* determines whether read, execute, write or some combination of accesses are permitted to the pages being mapped. To deny all access, set *prot* to *PROT_NONE*.Otherwise, specify permissions by an OR of *PROT_READ*, *PROT_EXECUTE*, and *PROT_WRITE*. A write access must fail if *PROT_WRITE* has not

been set, though the behavior of the write can be influenced by setting *MAP_PRIVATE* in the *flags* parameter, as described below.

The *flags* parameter provides other information about the handling of mapped pages:

- *MAP_SHARED* and *MAP_PRIVATE* specify the mapping type, and one of them must be specified. The mapping type describes the disposition of store operations made by "this" process into the address range defined by the mapping operation. If *MAP_SHARED* is specified, write references will modify the mapped object. No further operations on the object are necessary to effect a change — the act of storing into a *MAP_SHARED* mapping is equivalent to doing a *write* system call.

  On the other hand, if *MAP_PRIVATE* is specified, an initial write reference to a page in the mapped area will create a copy of that page and redirect the initial and successive write references to that copy. This operation is sometimes referred to as "copy-on-write" and occurs invisibly to the process causing the store. Only pages actually modified have copies made in this manner. *MAP_PRIVATE* mappings are used by system functions such as *exec(2)* when mapping files containing programs for execution. This permits operations by programs such as debuggers to modify the 'text' (code) of the program without affecting the file from which the program is obtained.

  The mapping type is retained across a *fork*.

  The private copy is not created until the first write; until then, other users who have the object mapped *MAP_SHARED* can change the object. That is, if one user has an object mapped *MAP_PRIVATE* and another user has the same object mapped *MAP_SHARED*, and the *MAP_SHARED* user changes the object before the *MAP_PRIVATE* user does the first write, then the changes appear in the *MAP_PRIVATE* user's copy that the system makes on the first write. If an application needs isolation from changes made by other processes, it should use *read* to make a copy of the data it wishes to keep isolated.
- *MAP_FIXED* informs the system that the value returned by *mmap* must be *addr*, exactly. The use of *MAP_FIXED* is discouraged, as it may prevent an implementation from making the most effective use of system resources. When *MAP_FIXED* is not set, the system uses *addr* as a hint to arrive at *paddr*. The *paddr* so chosen is an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. An *addr* value of zero grants the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken as a suggestion of a process address near which the mapping should be placed. When the system selects a value for *paddr*, it never places a mapping at address 0, nor replaces any extant mapping, nor maps into areas considered part of the potential data or stack 'segments'. The system strives to choose alignments for mappings that maximize the performance of the its hardware resources.

The file descriptor used in a *mmap* call need not be kept open after the mapping is established. If it is closed, the mapping will remain until such time as it is replaced by another call to *mmap* that explicitly specifies the addresses occupied by this mapping; or until the mapping is removed either by process termination or a call to *munmap*. Although the mapping endures independent of the existence of a file descriptor, changes to the file can influence accesses to the mapped area, even if they do not affect the mapping itself. For instance, should a file be shortened by a call to *truncate*, such that the mapping now 'overhangs' the end of the file, then accesses to that area of the file which 'does not exist' will result in SIGBUS signals. It is possible to create the mapping in the first place such that it 'overhangs' the end of the file — the only requirement when creating a mapping is that the addresses, lengths, and offsets specified in the operation be 'possible' (i.e., within the range permitted for the object in question), not that they exist at the time the mapping is created (or subsequently.)

Similarly, if a program accesses an address in a manner inconsistently with how it has been mapped (for instance, by attempting a store operation into a mapping that was established with only *PROT_READ* access), then a *SIGSEGV* signal will result. *SIGSEGV* signals will also result on any attempt to reference an address not defined by any mapping.

In general, if a program makes a reference to an address that is inconsistent with the mapping (or lack of a mapping) established at that address, the system will respond with a *SIGSEGV* violation. However, if a program makes a reference to an address consistent with how the address is mapped, but that address does not evaluate "at the time of the access" to allocated storage in the object being mapped, then the system will

respond with a *SIGBUS* violation. In this manner a program (or user) can distinguish between whether it is the mapping or the object that is inconsistent with the access, and take appropriate remedial action.

Using *mmap* to access system memory objects can simplify programs in a variety of ways. Keeping in mind that *mmap* can really be viewed as just a means to access memory objects, it is possible to program using *mmap* in many cases where you might program with *read* or *write*. However, it is important to realize that *mmap* can only be used to gain access to "memory" objects — those objects that can be thought of as randomly accessible storage. Thus, terminals and network connections cannot be accessed with *mmap* because they are not "memory". Magnetic tapes, even though they are memory devices, can not be accessed with *mmap* because storage locations on the tape can only be addressed sequentially. Some examples of situations which "can" be thought of as candidates for use of *mmap* over more traditional methods of file access include:

- Random access operations — either map the entire file into memory or, if the address space can not accommodate the file or if the file size is variable, create "windows" of mappings to the object.
- Efficiency — even in situations where access is sequential, if the object being accessed can be accessed via *mmap*, an efficiency gain may be obtained by avoiding the copying operations inherent in accesses via *read* or *write*.
- Structured storage — if the storage being accessed is collected as tables or data structures, algorithms can be more conveniently written if access to the file is treated just as though the tables were in memory. Previously, programs could not simply make storage or table alterations in memory and save them for access in subsequent runs; however, when the addresses of the table are defined by mappings to a file, then changes to the storage *are* changes to the file, and are thus automatically recorded in it.
- Scattered storage — if a program requires scattered regions of storage, such as multiple heaps or stack areas, such areas can be defined by mapping operations during program operation.

The remainder of this section will illustrate some other concepts surrounding mapping creation and use.

Mapping */dev/zero* gives the calling program a block of zero-filled virtual memory of the size specified in the call to *mmap*. */dev/zero* is a special device, that responds to *read* as an infinite source of bytes with the value 0, but when mapped creates an unnamed object to back the mapped region of memory. The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address of the system's choosing.

```
/*
*□Function□to□allocate□a□block□of□zeroed□storage.□□Parameter
*□is□the□number□of□bytes□desired.□□The□storage□is□mapped□as
*□MAP_SHARED,□so□that□if□a□fork□occurs,□the□child□process
*□will□be□able□to□access□and□modify□the□storage.□□If□we□wished
*□to□cause□the□child's□modifications□(as□well□as□those□by□the
*□parent)□to□be□invisible□to□the□ancestry□of□processes,□we
*□would□use□MAP_PRIVATE.
*/
caddr_t
get_zero_storage(int□len);
{
□□□□□int□fd;
□□□□□caddr_t□result;
□□□□□if□((fd□=□open("/dev/zero",□O_RDWR))□==□-1)
□□□□□□□□□□□□return□((caddr_t)-1);
□□□□□result□=□mmap(0,□len,□PROT_READ|PROT_WRITE,□MAP_SHARED,□fd,□0);
□□□□□(void)□close(fd);
□□□□□return□(result);
}
```

As written, this function permits a hierarchy of processes to use the area of allocated storage as a region of communication (for "implicit" interprocess communication purposes). Later in this chapter we will describe a set of system facilities that provide a similar function packaged for accomplishing the same purpose without

requiring that the processes be in a parent-child hierarchy.

In some cases, devices or files are "only" useful if accessed via mapping. An example of this is frame buffer devices used to support bit-mapped displays, where display management algorithms function best if they can operate randomly on the addresses of the display directly.

Finally, it is important to remember that mappings can be operated upon at the granularity of a single page. Even though a mapping operation may define multiple pages of an address space, there is "absolutely no restriction" that subsequent operations on those addresses must operate on the same number of pages. For instance, an *mmap* operation defining ten pages of an address space may be followed by subsequent *munmap* (see below) operations that remove every other page from the address space, leaving five mapped pages each followed by an unmapped page. Those unmapped pages may subsequently be mapped to different locations in the same or different objects, or the whole range of pages (or any partition, superset, or subset of the pages) used in other *mmap* or other memory management operations. Further, it must be noted that any mapping operation that operates on more than a single page can 'partially succeed' in that some parts of the address range can be affected even though the call returns a failure. Thus, an *mmap* operation that replaces another mapping, if it fails, may have deleted the previous mapping and failed to replace it. Similarly, other operations (unless specifically stated otherwise) may process some pages in the range successfully before operating on a page where the operation fails.

Not all device drivers support memory mapping. *mmap* fails if you try to map a device that does not support mapping.

3.1.3.2 Removing mappings

int
munmap(caddr_t addr, size_t len);

munmap removes all mappings for pages in the range [*addr*, *addr* + *len*) from the address space of the calling process. It is not an error to remove mappings from addresses that do not have them, and any mapping, no matter how it was established, can be removed with *munmap*. *munmap* does not in any way affect the objects that were mapped at those addresses.

3.1.3.3 Cache control

The Reliant UNIX memory management system can be thought of as a form of cache management, in which a processor's primary memory is used as a cache for pages from objects from the system's virtual memory. Thus, there are a number of operations which control or interrogate the status of this cache, as described in this section.

int
mincore(caddr_t addr, size_t len, char *vec);

mincore determines the residency of the memory pages in the address space covered by mappings in the range [*addr*, *addr* + *len*). Using the cache concept described earlier, this function can be viewed as an operation that interrogates the status of the cache, and returns an indication of what is currently resident in the cache. The status is returned as a char-per-page in the character array referenced by **vec* (which the system assumes to be large enough to encompass all the pages in the address range). Each character contains either a "1" (indicating that the page is resident in the system's primary storage), or a "0" (indicating that the page is not resident in primary storage.) The other 7 bits are reserved for possible future expansion. Other bits hold additional information. Therefore, programs testing residency should test only the least significant bit of each character.

mincore returns residency information that is accurate at an instant in time. Because the system may frequently adjust the set of pages in memory, this information may quickly be outdated. Only locked pages are guaranteed to remain in memory.

Various mapping control functions

int
memcntl(caddr_t□addr,□size_t□len,□int□cmd,□caddr_t□arg,□int□attr,□int mask);

memcntl provides several control operations over mappings in the range [*addr*, *addr* + *len*), including locking pages into physical memory, unlocking them, and writing pages to secondary storage. The functions

described in the rest of this section offer simplified interfaces to the *memcntl* operations.

```
int
mlock(caddr_t addr, size_t len);
int
munlock(caddr_t addr, size_t len);
```

mlock causes the pages referenced by the mapping in the range [*addr*, *addr* + *len*) to be locked in physical memory. References to those pages (through other mappings in this or other processes) will not result in page faults that require an I/O operation to obtain the data needed to satisfy the reference. Because this operation ties up physical system resources, and has the potential to disrupt normal system operation, use of this facility is restricted to the system administrator. The system will not permit more than a configuration-dependent limit of pages to be locked in memory simultaneously, the call to *mlock* will fail if this limit is exceeded.

munlock releases the locks on physical pages. Note that if multiple *mlock* calls are made through the same mapping, only a single *munlock* call will be required to release the locks (in other words, locks on a given mapping do not nest.) However, if different mappings to the same pages are processed with *mlock*, then the pages will not be unlocked until the locks on all the mappings are released.

Locks are also released when a mapping is removed, either through being replaced with an *mmap* operation or removed explicitly with *munmap*. A lock will be transferred between pages on the 'copy-on-write' event associated with a *MAP_PRIVATE* mapping, thus locks on an address range that includes *MAP_PRIVATE* mappings will be retained transparently along with the copy-on-write redirection (see *mmap* in Section "Creating and using mappings", for a discussion of this redirection).

Locking and unlocking pages in physical memory (mlockall, munlockall)

```
int
mlockall(int flags);
int
munlockall(void);
```

mlockall and *munlockall* are similar in purpose and restriction to *mlock* and *munlock*, except that they operate on entire address spaces. *mlockall* accepts a *flags* argument built as a bit-field of values from the set:

MCL_CURRENT
: Current mappings

MCL_FUTURE
: Future mappings

If *flags* is *MCL_CURRENT*, the lock is to affect everything currently in the address space. If *flags* is *MCL_FUTURE*, the lock is to affect everything added in the future. If *flags* is (*MCL_CURRENT* | *MCL_FUTURE*), the lock is to affect both current and future mappings.

munlockall removes all locks on all pages in the address space, whether established by *mlock* or *mlockall*.

Application program support (msync)

```
int
msync(caddr_t addr, size_t len, int flags);
```

msync supports applications which require assertions about the integrity of data in the storage backing their mapping, either for correctness or for coherent communications in a distributed environment. *msync* causes all modified copies of pages over the range [*addr*, *addr* + *len*) to be flushed to the objects mapped by those addresses. In the cache analogy discussed previously, *msync* is the cache 'write-back', or flush, operation. It is similar in purpose to the *fsync* operation for files.

msync optionally invalidates such cache entries so that further references to the pages cause the system to obtain them from their permanent storage locations.

The *flags* argument provides a bit-field of values that influences the behavior of *msync*. The bit names and their interpretations are:

MS_SYNC
: synchronized write

MS_ASYNC
: return immediately

MS_INVALIDATE
: invalidate caches

MS_SYNC causes *msync* to return only after all I/O operations are complete. *MS_ASYNC* causes *msync* to return immediately once all I/O operations are scheduled. *MS_INVALIDATE* causes all cached copies of data from mapped objects to be invalidated, requiring them to be reobtained from the object's storage upon the next reference.

3.1.3.4 Other mapping functions

Returning the system-dependent size of a memory page

```
long
sysconf(PAGESIZE);
```

sysconf returns the system-dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page, and instead should make use of *sysconf* to obtain that information. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set, increasing the importance of using this function for portability.

Assigning a protection mode to pages (mprotect)

```
int
mprotect(caddr_t addr, size_t len, int prot);
```

mprotect has the effect of assigning protection *prot* to all pages in the range [*addr*, *addr* + *len*). The protection assigned can not exceed the permissions allowed on the underlying object. For instance, a read-only mapping to a file that was opened for read-only access can not be set to be writable with *mprotect* (unless the mapping is of the *MAP_PRIVATE* type, in which case the write access is permitted since the writes will modify copies of pages from the object, and not the object itself).

3.1.4 Address space layout

Traditionally, the address space of a Reliant UNIX process has consisted of exactly three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process's stack. Text is read-only and shared, while the data and stack segments are private to the process.

Reliant UNIX 5.43 still uses text, data, and stack segments, though these should be thought of as constructs provided by the programming environment rather than by the operating system. As such, it is possible to construct processes that have multiple segments of each 'type', or of types of arbitrary semantic value - no longer are programs restricted to being built only from objects the system was capable of representing directly. For instance, a process's address space may contain multiple text and data segments, some belonging to specific programs and some shared among multiple programs. Text segments from shared libraries, for example, typically appear in the address spaces of many processes. A process's address space is simply a vector of pages, and there is no necessary division between different address-space segments. Process text and data spaces are simply groups of pages mapped in ways appropriate to the function they provide the program.

A process's address space is usually sparsely populated, with data and text pages intermingled. The precise mechanics of the management of stack space is machine-dependent. By convention, page 0 is not used. Process address spaces are often constructed through dynamic linking when a program is *exec*ed. Operations such as *exec* and dynamic linking build upon the mapping operations described previously. Dynamic linking is described further in [3].

While the system may have multiple areas that can be considered 'data' segments, for programming convenience the system maintains operations to operate on an area of storage associated with a process's initial heap storage area. A process can manipulate this area by calling *brk* and *sbrk*:

```
caddr_t
brk(caddr_t addr);
```

caddr_t
sbrk(int□incr);

brk sets the system's idea of the lowest data segment location not used by the caller to *addr* (rounded up to the next multiple of the system's page size).

sbrk, the alternate function, adds *incr* bytes to the caller's data space and returns a pointer to the start of the new data area.

3.2 Symbolic links

A symbolic link is a special type of file that represents another file. The data in a symbolic link consists of the path name of a file or directory to which the symbolic link file is linked. The link that is formed is called symbolic to distinguish it from a regular (also called a hard) link such as can be created by using the *ln(1)* command. A symbolic link differs functionally from a regular link in three major ways: files from different file systems may be linked together; directories as well as regular files may be symbolically linked by any user; and a symbolic link can be created even if the file it represents does not exist.

In order to understand how a symbolic link works, it is necessary to understand how the Reliant UNIX operating system views files. (The following description pertains to files that belong to the standard Reliant UNIX 5.43 file system type.) The internal representation of a file is contained in an inode, which contains a description of the layout of the file data on disk as well as information about the file, such as the file owner, the access permissions, and the access times. Every file has one inode, but a file may have several names, all of which point to the inode. Each name is called a regular (or hard) link.

When a file is created, an inode is allocated for it, the file contents are stored in data blocks, and an entry is created in a directory. A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file. The inode initially has a link count of one, which means that this file has one name (or one link to it).

We are now in a position to understand the difference between the creation of a regular and a symbolic link. When a user creates a regular link to a file with the *ln(1)* command, a new directory entry is created containing a new file name and the inode number of an existing file. The link count of the file is incremented.

In contrast, when a user creates a symbolic link both a new directory entry and a new inode are created. A data block is allocated to contain the path name of the file to which the symbolic link refers. The link count of the referenced file is not incremented.

Symbolic links can be used to solve a variety of common problems. For example, it frequently happens that a disk partition (such as root) runs out of disk space. With symbolic links, an administrator can create a link from a directory on that file system to a directory on another file system. Such a link provides extra disk space and is, in most cases, transparent to both users and programs.

Symbolic links can also help deal with the built-in path names that appear in the code of many commands. Changing the path names would require changing the programs and recompiling them. With symbolic links, the path names can effectively be changed by making the original files symbolic links that point to new files.

Symbolic links can also be very useful in a shared resource environment. For example, if it is important to have a single copy of certain administrative files, symbolic links can be used to help share them. Symbolic links can also be used to share resources selectively. Suppose a system administrator wants to do a remote mount of a directory that contains sharable devices. These devices must be in */dev* on the client system, but this system has devices of its own so the administrator does not want to mount the directory onto */dev*. Rather than do this, the administrator can mount the directory at a location other than */dev* and then use symbolic links in the */dev* directory to refer to these remote devices. (This is similar to the problem of built-in path names since it is normally assumed that devices reside in the */dev* directory.)

Finally, symbolic links can be valuable within the context of the virtual file system (VFS) architecture. With VFS new services, such as higher performance files, events, and network IPC, may be provided on a file system basis. Symbolic links can be used to link these services to home directories or to places that make more sense to the application or user. Thus one might create a database index file in a RAM-based file system type and symbolically link it to the place where the database server expects it and manages it.

3.2.1 Using symbolic links

The phrases "following symbolic links" and "not following symbolic links" as they are used in this document refer to the evaluation of the last component of a path name. In the evaluation of a path name, if any component other than the last is a symbolic link, the symbolic link is followed and the referenced file is used in the path name evaluation. However, if the last component of a path name is a symbolic link, the link may or may not be followed.

3.2.1.1 Properties of symbolic links

This section summarizes some of the essential characteristics of symbolic links. Succeeding sections describe how symbolic links may be used, based on the charac-teristics outlined here.

As we have seen above, a symbolic link is a new type of file that represents another file. The file to which it refers may be of any type; a regular file, a directory, a character-special, block-special, or FIFO-special file, or another symbolic link. The file may be on the local system or on a remote system. In fact, the file to which a symbolic link refers does not even have to exist. In particular, the file does not have to exist when the symbolic link is created or when it is removed.

Creation and removal of a symbolic link follow the same rules that apply to any file. To do either, the user must have write permission in the directory that contains the symbolic link.

The ownership and the access permissions (mode) of the symbolic link are ignored for all accesses of the symbolic link. It is the ownership and access permissions of the referenced file that are used. A symbolic link cannot be opened or closed and its contents cannot be changed once it has been created.

If the file */usr/jan/junk* is a symbolic link to the file */etc/passwd*, in effect the file name */etc/passwd* is substituted for junk so that when the user executes

cat□/usr/jan/junk

it is the contents of the file */etc/passwd* that are printed.

Similarly, if */usr/jan/junk* is a symbolic link to the file *../junk2*, executing

cat□/usr/jan/junk
is□the□same□as□executing
cat□/usr/jan/../junk2

or

cat /usr/junk2

When a symbolic link is followed and brings a user to a different part of the file tree, we may distinguish between where the user really is (the physical path) and how the user got there (the virtual path). The behavior of */usr/bin/pwd*, the shell built-in *pwd*, and . . are all based on the physical path. In practical terms this means that there is no way for the user to retrace the path which brought the user to the current position in the file tree.

Other shells may use the virtual path. For example, by default the Korn shell *pwd* uses the virtual path, though there is an option allowing the user to make it use the physical path.

Consider the case shown in this figure, where */usr/include/sys* is a symbolic link to */usr/src/uts/sys*.

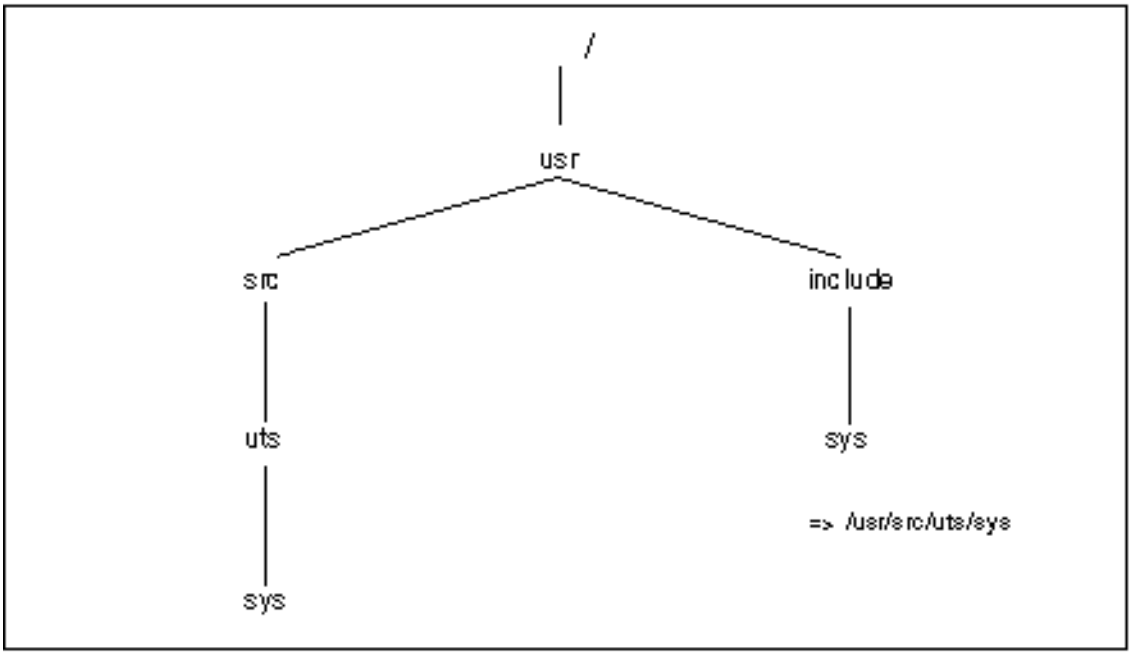


Figure 5: File tree with symbolic link

Here if a user enters *cd /usr/include/sys* and then enters *pwd*, the result is */usr/src/uts/sys*. If the user then enters *cd* .. followed by *pwd*, the result is */usr/src/uts*.

3.2.1.2 Creating symbolic links

Syntax and semantics

To create a symbolic link, the new system call *symlink(2)* is used and the owner must have write permission in the directory where the link will reside. The file is created with the user's user-id and group-id but these are subsequently ignored. The mode of the file is created as 0777.

No checking is done when a symbolic link is created. There is nothing to stop a user from creating a symbolic link that refers to itself or to an ancestor of itself or several links that loop around among themselves. Therefore, when evaluating a path name, it is important to put a limit on the number of symbolic links that may be encountered in case the evaluation encounters a loop. The variable *MAXSYMLINKS* is used to force the error *ELOOP* after *MAXSYMLINKS* symbolic links have been encountered. The value of *MAXSYMLINKS* should be at least 20.

To create a symbolic link, the *ln(1)* command is used with the –s option. If the –s option is not used and a user tries to create a link to a file on another file system, a symbolic link will not be created and the command will fail.

The syntax for creating symbolic links is as follows:

ln –s sourcefile1 [sourcefile2 ...] target

With two arguments:

- For each *sourcefile*, a file is created in *target* whose name is *sourcefile* or its last component (`*basename sourcefile*`) and is a symbolic link to *sourcefile*.
- If *target* is not an existing directory, an error is returned.
- Each *sourcefile* and *target* may reside on different file systems.
- *sourcefile1* may be any path name and need not exist.
- *target* may be an existing directory or a non-existent file.

- If *target* is an existing directory, a file is created in directory *target* whose name is the last component of *sourcefile1* (`*basename sourcefile1*`). This file is a symbolic link that references *sourcefile1*.
- If *target* does not exist, a file with name *target* is created and it is a symbolic link that references *sourcefile1*.
- If *target* already exists and is not a directory, an error is returned.
- *sourcefile1* and *target* may reside on different file systems.

  With more than two arguments:

Examples

The following examples show how symbolic links may be created.

ln -s /usr/src/uts/sys /usr/include/sys

In this example */usr/include* is an existing directory. But the file *sys* does not exist so it will be created as a symbolic link that refers to */usr/src/uts/sys*. The result is that when the file */usr/include/sys/x* is accessed, the file */usr/src/uts/sys/x* will actually be accessed.

This kind of symbolic link may be used when files exist in the directory */usr/src/uts/sys* but programs often refer to files in */usr/include/sys*. Rather than creating corresponding files in */usr/include/sys* that are hard links to files in */usr/src/uts/sys*, one symbolic link can be used to link the two directories. In this example */usr/include/sys* becomes a symbolic link that links the former */usr/include/sys* directory to the */usr/src/uts/sys* directory.

ln□-s□□/etc/group□□.

In this example the *target* is a directory (the current directory), so a file called *group* (`*basename /etc/group*`) is created in the current directory that is a symbolic link to */etc/group*.

ln□-s□□/fs1/jan/abc□□/var/spool/abc

In this example we imagine that */fs1/jan/abc* does not exist at the time the command is issued. Nevertheless, the file */var/spool/abc* is created as a symbolic link to */fs1/jan/abc*. Later, */fs1/jan/abc* may be created as a directory, regular file, or any other file type.

The following example illustrates the use of more than two arguments:

ln□-s□□/etc/group□□/etc/passwd□□.

The user would like to have the *group* and *passwd* files in the current directory but cannot use hard links because */etc* is a different file system. When more than two arguments are used, the last argument must be a directory; here it is the current directory. Two files, *group* and *passwd*, are created in the current directory, each a symbolic link to the associated file in */etc*.

3.2.1.3 Removing symbolic links

Normally, when accessing a symbolic link, one follows the link and actually accesses the referenced file. However, this is not the case when one attempts to remove a symbolic link. When the *rm(1)* command is executed and the argument is a symbolic link, it is the symbolic link that is removed; the referenced file is not touched.

3.2.1.4 Accessing symbolic links

Suppose *abc* is a symbolic link to file *def*. When a user accesses the symbolic link *abc*, it is the file permissions (ownership and access) of file *def* that are actually used; the permissions of *abc* are always ignored. If file *def* is not accessible (i.e., either it does not exist or it exists but is not accessible to the user because of access permissions) and a user tries to access the symbolic link *abc*, the error message will refer to *abc*, not file *def*.

3.2.1.5 Copying symbolic links

This section describes the behavior of the *cp(1)* command when one or more arguments are symbolic links. With the *cp(1)* command, if any argument is a symbolic link, that link is followed. Then the semantics of the command are as described in [4]. Suppose the command line is

cp sym file3

where *sym* is a symbolic link that references a regular file *test1* and *file3* is a regular file.

After execution of the command, *file3* gets overwritten with the contents of the file *test1*. If the last argument is

a symbolic link that references a directory, then files are copied to that directory. Suppose the command line is

cp file1 sym symd

where *file1* is a regular file, *sym* is a symbolic link that references a regular file *test1*, and *symd* is a symbolic link that references a directory *DIR*. After execution of the command, there will be two new files, *DIR/file1* and *DIR/sym* that have the same contents as *file1* and *test1*.

3.2.1.6 Linking symbolic links

This section describes the behavior of the *ln(1)* command when one or more arguments are symbolic links. To understand the difference in behavior between this and the *cp(1)* command, it is useful to think of a copy operation as dealing with the contents of a file while the link operation deals with the name of a file.

If the first argument to *ln(1)* is a symbolic link it is not followed, and a hard link is made to the symbolic link. With the last argument, a *stat(2)* is done to see if it is a directory; if it is, files are linked in that directory. Otherwise, if the last argument is an existing file, it is overwritten. This means that if the last argument is a symbolic link to a directory, it is followed but if it is a symbolic link to a regular file, the symbolic link is overwritten.

For example, if the command line is

ln sym file1

where *sym* is a symbolic link that references a regular file *foo*, and *file1* is a regular file, *file1* is overwritten and hard-linked to *sym*, i.e., *file1* becomes a symbolic link that references *foo*. Thus a hard link has been created to a symbolic link.

If the command is

ln file1 sym

where the files are the same as in the first example, *sym* is overwritten and hard-linked to *file1*.

When the last argument is a directory as in

ln file1 sym symd

where *symd* is a symbolic link to a directory *DIR*, the file *DIR/file1* is hard-linked to *file1* and *DIR/sym* is hard-linked to *sym*.

3.2.1.7 Moving symbolic links

This section describes the behavior of the *mv(1)* command. Like the *ln(1)* command, *mv(1)* deals with file names rather than file contents. With two arguments, a user invokes the *mv(1)* command to rename a file. Therefore, one would not want to follow the first argument if it is a symbolic link because it is the name of the file that is to be changed rather than the file contents. Suppose that *sym* is a symbolic link to */etc/passwd* and *abc* is a regular file. If the command

mv sym abc

is executed, the file *sym* is renamed *abc* and is still a symbolic link to */etc/passwd*. If *abc* existed (as a regular file or a symbolic link to a regular file) before the command was executed, it is overwritten.

Suppose the command is

mv sym1 file1 symd

where *sym1* is a symbolic link to a regular file *foo*, *file1* is a regular file, and *symd* is a symbolic link that references a directory *DIR*. When the command is executed, the files *sym1* and *file1* are moved from the current directory to the *DIR* directory so that there are two new files, *DIR/sym1*, which is still a symbolic link to *foo*, and *DIR/file1*.

In Reliant UNIX 5.43, the *rename(2)* system call is used by the *mv(1)* command. If the first argument to *rename(2)* is a symbolic link, *rename(2)* does not follow it; instead it renames the symbolic link itself. Prior to Reliant UNIX 5.43 a file was moved using the *link(2)* system call followed by the *unlink(2)* system call. Since *link(2)* and *unlink(2)* do not follow symbolic links, the result of those two operations is the same as the result of a call to *rename(2)*.

3.2.1.8 Archiving commands

The *cpio(1)* command is used to copy file archives usually to or from a storage medium such as a tape, disk, or diskette. By default, *cpio(1)* does not follow symbolic links. However, a new *-L* option may be used with the *-o* and *-p* options to indicate that symbolic links should be followed. Note that this option is not valid with the *-i* option.

Normally, a user invokes the *find(1)* command to produce a list of filenames and pipes this into the *cpio(1)* command to create an archive of the files listed. The *find(1)* command also has a new option *-follow* to indicate that symbolic links should be followed. If a user invokes *find(1)* with the *-follow* option, then *cpio(1)* must also be invoked with its new option *-L* to indicate that it too should follow symbolic links.

When evaluating the output from *find(1)*, following or not following symbolic links only makes a difference when a symbolic link to a directory is encountered. For example, if */usr/jan/symd* is a symbolic link to the directory *../joe/test* and files *test1* and *test2* are in directory */usr/joe/test*, the output of a *find* command starting from */usr/jan* will include the file */usr/jan/symd* if symbolic links are not followed but will include the files */usr/jan/symd*, */usr/jan/symd/test1*, and */usr/jan/syd/test2* when symbolic links are followed.

If the user wants to preserve the structure of the directories being archived, it is recommended that symbolic links not be followed on both commands. (This is the default.) When this is done symbolic links will be preserved and the directory hierarchy will be duplicated as it was.

If the user is more concerned that the contents of the files be saved, then the user should use the *-L* option to *cpio(1)* and the *-follow* option to *find(1)* to follow symbolic links.

The user should take care not to mix modes, that is, the user should either follow or not follow symbolic links for both *cpio(1)* and *find(1)*. If modes are mixed, an archive will be created but the resulting hierarchy created by *cpio -i* may exhibit unexpected and undesirable results.

When copying in using the *–i* option to *cpio(1)*, symbolic links will be copied as is. It should be noted that systems prior to Reliant UNIX 5.43 do not understand symbolic links and the result of copying in a symbolic link will be a regular file whose contents are the path name of the referenced file. So if a user is creating an archive to be read in on a pre-Reliant UNIX 5.43 system, it may be more useful to follow symbolic links.

3.2.1.9 File ownership and permissions

The commands *chmod(1)*, *chown(1)*, and *chgrp(1)*, and their corresponding system calls are used to change the mode and ownership of a file. If the argument to *chmod(1)*, *chown(1)*, or *chgrp(1)* is a symbolic link, the mode and ownership of the referenced file rather than of the symbolic link itself will be changed. In such cases, the link is followed.

Once a symbolic link has been created, its permissions cannot be changed. By default, the *chown(1)* and *chgrp(1)* commands change the owner and group of the referenced file. However, a new *-h* option enables the user to change the owner and group of the symbolic link itself. This is useful for removing files from sticky directories.

3.2.1.10 Using symbolic links with NFS

To use symbolic links on two systems running NFS, both systems must be running Reliant UNIX 5.43. In cases where the server is a Reliant UNIX 5.43system but the client is not, errors will be generated when the client encounters a symbolic link.

When using symbolic links in an NFS environment, it is important to understand how pathnames are evaluated. The rule by which evaluations are performed is simple. Symbolic links that a client encounters on the server are interpreted in accordance with the client's view of the file tree.

Users on a server system must keep this rule in mind when they create symbolic links in order to avoid problems. The examples that follow illustrate situations in which failure to consider the client's view of the file tree can lead to problems.

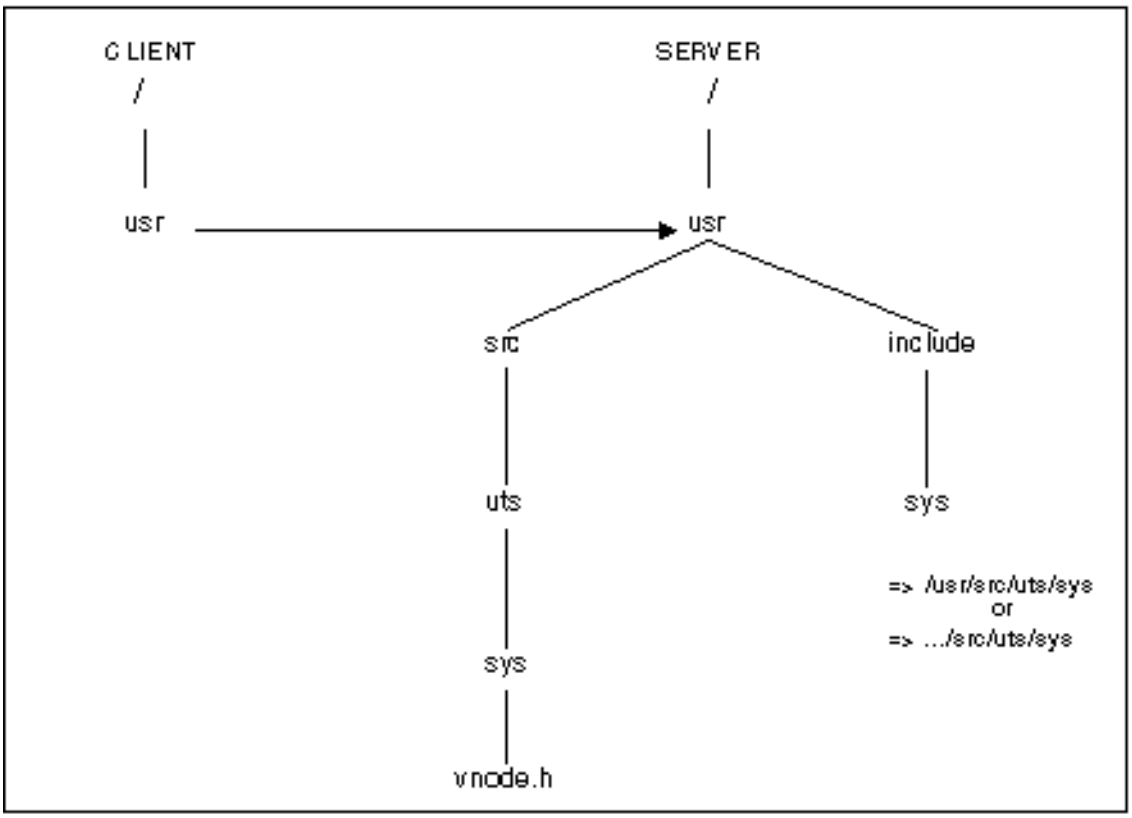


Figure 6: Symbolic links with NFS: Example 1

In the example shown in this figure, the server advertises its */usr* file system as USR. If the server creates the symbolic link */usr/include/sys* as an absolute pathname to */usr/src/uts/sys*, evaluation of the link will work as intended as long as a client mounts USR as */usr*. Another way of saying this is that if the file tree naming conventions are the same on the client and the server, things will work as intended

However, if the client mounts USR as */mnt/usr*, when the symbolic link */usr/src/uts/sys* is evaluated, the evaluation will be done with respect to the client's view of the file tree and will not cross the mount point back to the server but will remain on the client. Thus the client will not access the file intended.

In this situation the server should create the symbolic link as a relative path name, *../src/uts/sys*, so that evaluation will produce the desired results regardless of where the client mounts USR.

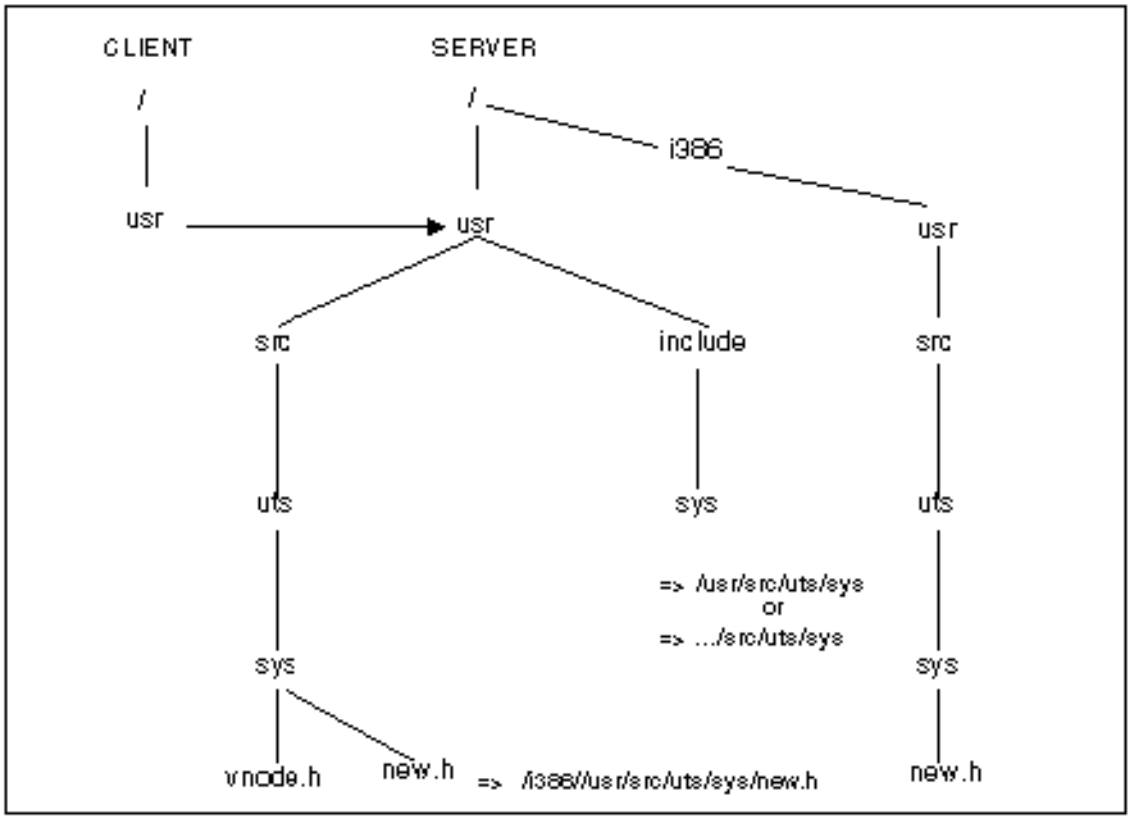


Figure 7: Symbolic links with NFS: Example 2

Another example of a situation that could cause problems is shown in this figure. Here again the server advertises its */usr* file system as USR. But in this case the server has a symbolic link from */usr/src/uts/sys/new.h* to */i386/usr/src/uts/sys/new.h*.

Because the referenced file, */i386/usr/src/uts/sys/new.h*, is outside of the advertised resource, users on the server can access this file but users on the client cannot. In this example, it would make no difference if the symbolic link was a relative rather than an absolute pathname, because the directory */i386* on the server is not part of the client's name space. When the symbolic link is evaluated, the system will look for the file on the client and will not follow the link as intended.

3.3 File and device I/O

The lowest level of I/O under Reliant UNIX 5.43 does not provide buffering or any other such services, but it does offer the best event control facilities. System calls acting directly on the Reliant UNIX 5.43 kernel control user I/O. Under Reliant UNIX 5.43 the I/O system calls are kept straightforward and uniform to neutralize the differences between files, devices and access modes. The same read and write system calls apply to simple disk files and also to I/O devices such as terminals, tape drives and line printers. They make no distinction between 'random' and 'sequential' I/O; and they do not define a logical record size for files. Communication between programs and peripherals is handled by a single uniform interface. Programmers do not need to identify the devices in their program code; the device names can be specified at the time of program execution.

All I/O takes the form of reading from or writing to files, as all I/O peripherals, including the user's terminal, are represented by files in the file system. For each supported device there is an entry in the file system hierarchy. Thus device names have the same format as file names, and devices and files are subject to the sane protection mechanisms.

For an I/O device, a file is an ordered set of data bytes. The size of an input file is governed by an end-of-file

condition depending on device-specific attributes. The size of a regular file is governed by the position and number of bytes in the file. It is neither necessary nor possible to define the file size in advance.

In addition to the devices which are traditionally available, there are names for disk devices viewed as physical units outside the file system and for absolutely addressed memory. In practice the most important device is the user's terminal. Uniform handling of communication devices and other files by the same I/O calls simplifies the task of redirecting command input and output from the terminal to another file. Of course there are some unavoidable differences. Thus Reliant UNIX 5.43 usually processes terminal input a line at a time, as character and line deletion cannot be completed until a line has been entered in full. Programs which attempt to read a greater number of bytes from a terminal have to wait until a whole line has been entered. They may then be informed that a smaller number of bytes has been read. All programs must anyway be prepared for this eventuality, as a read operation on a disk file returns fewer bytes than were requested if end-of-file is encountered. Terminal reads are generally fully compatible with disk file reads.

3.3.1 File descriptors

The Reliant UNIX 5.43 file and device I/O functions identify a file by means of a small positive integer value known as the *file descriptor*, which is declared as follows:

int fildes

where *fildes* stands for the file descriptor. The file descriptor identifies an open file that data can be read from or written to. Reliant UNIX 5.43 manages all information relating to an open file. The user program always refers to the file by way of its descriptor. All I/O for that file uses the file descriptor instead of the file name to identify the file.

A number of file descriptors may refer to the same file. Each descriptor is assigned I/O information for the associated file:

- a file pointer to specify which byte of the file is the next for reading or writing,
- file status and access mode (*read*, *write*, *read/write*, see *open*(2)),
- the "close-on-exec" flag (see *fcntl*(2)).

I/O at the user's terminal occurs so frequently that special arrangments allow the mechanism to be simplified. When the command interpreter (the shell) runs a program, it opens three files (*standard input*, *standard output* and *standard error*) with the file descriptors *0*, *1* and *2* respectively. They are all usually associated with the terminal. Thus a program which reads from file descriptor *0* and writes to file descriptors *1* and *2* can perform terminal I/O without having to open any files. When I/O is redirected to and from files using < and > as in

prog <in_file >out_file

the shell alters the standard allocations of file descriptors *0* and *1* from the terminal to the specified files. Similar arrangements apply to I/O on interprocess channels. File descriptor *2* usually remains associated with the terminal so that error messages are displayed there. It is always the shell that does the redirection, not the program. The program does not need to know where output is going to as long as file descriptor *0* is used for input and file descriptors *1* and *2* for output.

3.3.2 File read and write operations

The *read* and *write* functions handle file I/O. The first argument to both functions is a file descriptor. The second is a buffer in the user program that data is read to or written from. The third argument is the number of bytes to read or write. The call returns the number of bytes actually read or written. The call formats are as follows:

n = read(fildes, buffer, number);
n = write(fildes, buffer, number);

Up to *number* bytes are transferred between the file identified by *fildes* and the array pointed to by *buffer*. The return value *n* stands for the number of bytes actually transferred.

With write operations, the return value is the number of bytes actually written. It is generally an error for this value to be different from the requested number of bytes. With *write*, *n* is equal to *number* except in exceptional circumstances (such I/O errors or the end of a physical medium in the case of device special files). With *read*,

it is not necessarily an error for *n* to be smaller than *number*.

With read operations, the number of bytes returned may be smaller that the number requested because there are no longer *number* bytes left to read. If the file pointer is so close to the end of the file that reading *number* characters would result in reading beyond end-of-file (EOF), the number of bytes read is restricted to the number remaining up to EOF. Typewriter-style terminals never return more than one line of input. (If the file is a terminal, *read* normally reads only as far as the next newline character, which will generally be less than was requested.)

If a *read* call returns with an *n* of zero, EOF has been reached. This happens with disk files if the file offset is equal to the current file size. It is possible to generate an EOF from a terminal by entering a device-dependent escape sequence. The *read* function returns *0* on EOF and *-1* on error.

The number of bytes to be read or written is an arbitrary number. The two commonest values are *1*, i.e. one character at a time ("raw" or "unbuffered"), and *512* (which is the physical block size on many peripherals). The latter value is the most efficient; but I/O which handles one character at a time is not too inefficient, either. Written bytes affect only that part of a file which is delimited by the position of the file offset and the *number*. No other parts of the file are modified. If the last byte is located beyond EOF, the file is extended as necessary.

A simple program using the *read* and *write* functions to copy its input to its output can perform all the copy operations because input and output can be redirected to all files or devices.

Example of read and write

```
#define□□BUFSIZE□□512
main()□□□/*□copy□input□to□output□*/
{
□□□char□buf[BUFSIZE];
□□□int□□n;
□□□while□((n□=□read(0,□buf,□BUFSIZE))□>□0)
□□□□□□write(□1,□buf,□n);
□□□exit(0);
}
```

If the file size is not a multiple of *BUFSIZE*, one invocation of *read* will return a smaller number of bytes for writing by *write*. The next invocation of *read* will return zero, thus indicating that EOF has been reached.

To illustrate how *read* and *write* can be used to create higher-level functions (like *getchar* and *putchar*), the following example demonstrates *getchar*, which reads its input unbuffered:

```
#define□□CMASK□□□0377□□/*□for□making□char's□>□0□*/
getchar()□/*□unbuffered□single□character□input□*/
{
□□□char□c;
□□□return((read(0,□&c,□1)□>□0)□?□c□&□CMASK□:□EOF);
}
```

The *c* variable must be declared as *char* because *read* expects a pointer to a character. The character returned must be masked against *0377* to ensure that it is a positive character. Otherwise it might be made negative by an additional sign.

The second version of *getchar* which follows handles input in large blocks and output a character at a time.

```
#define□□CMASK□□□0377□□/*□for□making□char's□>□0□*/
#define□□BUFSIZE□□512
getchar()□□/*□buffered□version□*/
{
□□□static□char□□□□buf[BUFSIZE];
□□□static□char□□□□*bufp□=□buf;
□□□static□int□□□□□n□=□0;
□□□if□(n□==□0)□□{□□□/*□buffer□is□empty□*/
```

```
□□□□□□n□=□read(0,□buf,□BUFSIZE);
□□□□□□bufp□=□buf;
□□□}
□□□return((--n□>=□0)□?□*bufp++□&□CMASK□:□EOF);
}
```

3.3.3 Opening, creating and closing files

With the exception of standard input, standard output and standard error, any files that are to be read from or written must be opened explicitly. There are two functions available for this purpose: *open* and *creat* (see *open*(2) and *creat*(2) in [9]). For reads and writes on a file which is assumed to exist, the file must be opened with the following call:

fildes = open(name, oflag);

The *name* argument is a string in the form of a path name in the Reliant UNIX 5.43 file system. The *oflag* argument whether to read from the file, write to it or 'update' it (both read and write). The return value *fildes* is a file descriptor used to identify the file in subsequent file handling operations.

The *open* is similar to the *fopen* in the standard I/O library. However, *open* does not return a pointer to *FILE*: it returns a file descriptor in the form of an *int* value (see *fopen*(3S) and *stdio*(3S) in [9]). The values of the access mode argument (*oflag*) are different as well (these flags are given in the */usr/include/fcntl.h* header file):

O_RDONLY
: for reading only.

O_WRONLY
: for writing only.

O_RDWR
: for reading and writing.

The *open* function returns *-1* on error. Otherwise it return a valid file descriptor for an open file.

An attempt to *open* a file which does not exist results in an error. To create new files or overwrite old files the *creat* function is used. The *creat* system call creates the specified file provided it does not yet exist. It it already exists, its length is truncated to zero. *creat* opens the new file for writing as well and, like *open*, returns a file descriptor. The format if the *creat* call is as follows:

fildes = creat(name, pmode);

This call returns a file descriptor if the file *name* has been created. Otherwise it returns *-1*. An attempt to use *creat* to create a file which already exists does not result in an error; but the file's length is truncated to zero.

If the file is new, *creat* creates it with the *permission mode* defined in the *pmode* argument. The Reliant UNIX 5.43 file system assigns a file nine permission bits to control *read*, *write* and *execute permission* for the file's *owner*, the owner's *group* and all *others*. The access permissions are specified in the form of a three-digit octal number. Thus the number *0755* indicates *read*, *write* and *execute permission* for the *owner* and *read* and *execute permission* for the *group* and all *others*.

As an illustration let us use a simplified version of the Reliant UNIX 5.43 *cp* utility (which copies one file to another):

```
#define□□NULL□0
#define□□BUFSIZE□512
#define□□PMODE□0644□/*□RW□owner,□R□group□&□others□*/
main(argc,□argv)□□□□/*□cp:□copy□f1□to□f2□*/
□□□int□argc;
□□□char□*argv[□];
{
□□□int□□f1,□f2,□n;
□□□char□buf[BUFSIZE];
□□□if□(argc□!=□3)
```

```
□□□□□□error("Usage:□cp□from□to",□NULL);
□□□if□((f1□=□open(argv[1],□0))□==□-1)
□□□□□□error("cp:□can't□open□%s",□argv[1]);
□□□if□((f2□=□creat(argv[2],□PMODE))□==□-1)
□□□□□□error("cp:□can't□create□%s",□argv[2]);
□□□while□((n□=□read(f1,□buf,□BUFSIZE))□>□0)
□□□□□□if□(write(f2,□buf,□n)□!=□n)
□□□□□□□□□□error("cp:□write□error",□NULL);
□□□exit(0);
}
error(s1,□s2)□□/*□print□error□message□and□die□*/
□□□char□*s1,□*s2;
{
□□□printf(s1,□s2);
□□□printf("\n")
□□□exit(1);
}
```

The essential nature of the simplification is that this version can only copy one file and does not accept a directory as its second argument.

There a ceiling, *OPEN_MAX*, on the number of files that can be open for a process at one time. So a program which processes a large number of files must allow for reuse of file descriptors. The *close* function breaks the connection between a file descriptor and an open file, thereby relasing the file descriptor for use with another file. On exit from a program (*exit*) or on return from the main program, all open files are closed.

3.3.4 Random access - lseek

File I/O is usually performed sequentially: each *read* or *write* operation starts immediately after the point in the file at which the previous one ended. Thus if a particular byte in the file was the last to be written (or read), the next I/O call implicitly operates on the one that comes immediately after it. For each open file Reliant UNIX 5.43 maintains a file pointer marking the next byte to be read or written. If *n* bytes are read or written the file pointer is moved by *n* bytes. If necessary, though, it is also possible to do reads and writes on a file in random order. *lseek* allows you to move around in a file without reading or writing. Random (or direct access) I/O simply involves using the *lseek* call to move the file pointer to the appropriate position in the file. The format for calling *lseek* is as follows:

lseek(fildes, offset, whence);

or

location = lseek(fildes, offset, whence);

This call moves the pointer in the file identified by file descriptor *fildes* by *offset* bytes from the starting point identified by *whence*, which is either the start of the file, the current pointer location, or the end of the file. The next read or write then applies to the new pointer location. *offset* may be negative. *lseek* cannot be applied to certain devices (such as paper tape devices and terminals) in such cases is simply ignored. The value of *location* corresponds to the displacement of the file pointer from the start of the file. The *offset* argument is of type *off_t*, which is defined as *long* in the *<types.h>* header file. *fildes* and *whence* are *int* values. The *whence* argument can be *SEEK_SET*, *SEEK_CUR* or *SEEK_END*, indicating that *offset* is calculated from the start of the file, the current pointer location, or the end of the file respectively. If, for example, you want to append data to the end of a file, you first seek to the end of the file:

lseek(fildes, 0L, SEEK_END);

If you want to go back to the beginning ('rewind'):

lseek(fildes, 0L, SEEK_SET);

Note the *0L* argument. It could also be written in the form *(long) 0*.

Using *lseek* you can treat files more or less as large arrays, admittedly at a loss of access speed. The following simple function, for example, reads a random number of bytes from an arbitrary position in a file:

```
get(fd,□p,□buf,□n)□/*□read□n□bytes□from□position□p□*/
□□□int□fd,□n;
□□□long□p;
□□□char□*buf;
{
□□□lseek(fd,□p,□SEEK_SET);□□/*□move□to□p□*/
□□□return(read(fd,□buf,□n));
}
```

3.3.5 Interprocess channels

The *pipe* system call creates an *interprocess channel* (pipe). This is a type of unnamed FIFO (First In First Out) file which acts as an I/O channel between two processes. One process writes to the process channel, the other reads from it. Most interprocess channels are created by the shell. For example:

ls | pr

This call connects the standard output of *ls* with the standard input of *pr*. However, it is sometimes more practical for a process to set up this sort of connection for itself. This section describes the creation and use of interprocess channel connections.

As an interprocess channel is used both for reading and for writing, *pipe* returns two file descriptors:

```
int□□fd[2];
stat□=□pipe(fd);
if□(stat□==□-1)
□□□□/*□there□was□an□error□...□*/
```

Here *fd* stands for an array containing two file descriptors: *fd[0]* for the read end and *fd[1]* for the write end. These file descriptors can be uused in *read*, *write* and *close* calls just like any other file descriptor.

Interprocess channel implementation takes the form of implicit *lseek* operations preceding each call to *read* or *write* to implement the "first in, first out" principle. The system handles data buffering and process synchronization in order to keep the relationship between the reading process and the writing process in balance. A process reading from an empty interprocess channel waits until data arrives. A process writing to a full interprocess channel waits until the channel has grown a little emptier. If the write end is closed, any subsequent call to read will be sent an end-of-file condition.

To illustrate interprocess channel operation, consider a function *popen(cmd, mode)* which creates the process *cmd*, returns a file descriptor and performs a read or a write as specified in *mode*:

fout = popen("pr", WRITE);

This call creates a process which executes the *pr* command. Subsequent calls to *write* with file descriptor *fout* send data to this process through the interprocess channel.

Example of interprocess channel operation

```
#include□<stdio.h>
#define□□□READ□□□□0
#define□□□WRITE□□1
#define□□□tst(a,□b)□(mode□==□READ□?□(b)□:□(a))
static□□□□int□popen_pid;
popen(cmd,□mode)
□□□char□*cmd;
□□□int□□mode;
{
□□□int□p[2];
□□□if□(pipe(p)□<□0)
□□□□□□return(NULL);
□□□if□((popen_pid□=□fork(□))□==□0)□{
□□□□□□close(tst(p[WRITE],□p[READ]));
```

```
□□□□□□close(tst(0,□1));
□□□□□□dup(tst(p[READ],□p[WRITE]));
□□□□□□close(tst(p[READ],□p[WRITE]));
□□□□□□execl("/bin/sh",□"sh",□"-c",□cmd,□0);
□□□□□□_exit(1)□/*□disaster□occurred□if□we□got□here□*/
□□□}
□□□if□(popen_pid□==□-1)
□□□□□□return(NULL);
□□□close(tst(p[READ],□p[WRITE]));
□□□return(tst(p[WRITE],□p[READ]));
}
```

The *popen* function first calls *pipe* to create an interprocess channel. It then calls *fork*, thereby creating a new process. The child process checks the mode (read or write) closes the other end of the interprocess channel and then (using *execl*) calls the shell to allow the required process to run. The parent process likewise closes the end of the interprocess channel that it does not need. These *close* operations are needed to allow for end-of-file checking. For example, if a child process that wants to read does not close the write end of the interprocess channel, it will never get the EOF signal from the channel as there is a potentially active writing process. The sequence of *close* operations in the child process is not quite trivial. Assume we want to create a child process to read data from the parent process. The first *close* closes the write end of the interprocess channel. The reads end stays open.

The following example illustrates the usual way of assigning the interprocess channel's descriptor to the standard input of the child process:

```
close(tst(0,□1));
dup(tst(p[READ],□p[WRITE]));
```

The *close* call closes file descriptor *0*, the standard input. The *dup* system call duplicates the descriptor of an open file. File descriptors are assigned in ascending order. *dup* returns the first available file descriptor. Thus the effect of *dup* here is to copy the file descriptor for the interprocess channel (read end) to file descriptor *0*. This turns the read end of the interprocess channel into the standard input. (This is not quite straightforward, but it is a common trick.) Then the old read end of the interprocess channel is closed. A comparable sequence of operations allows the child process to write to the parent process.

The operation is not complete until a *pclose* function closes the interprocess channel opened by *popen*. The main reason for using a separate function to do this rather than *close* is that it is best to wait for the child process to terminate.

```
#include□<signal.h>
pclose(fd)□□□/*□close□pipe□descriptor□*/
□□□int□fd;
{
□□□struct□sigaction□o_act,□h_act,□i_act,□q_act;
□□□extern□pid_t□popen_pid;
□□□pid_t□c_pid;
□□□int□□□c_stat;
□□□close(fd);
□□□sigaction(SIGINT,□SIG_IGN,□&i_act);
□□□sigaction(SIGQUIT,□SIG_IGN,□&q_act);
□□□sigaction(SIGHUP,□SIG_IGN,□&h_act);
□□□while□((c_pid=wait(&c_stat))!=-1□&&□c_pid!=popen_pid);
□□□if□(c_pid□==□-1)
□□□□□□c_stat□=□-1;
□□□sigaction(SIGINT,□&i_act,□&o_act);
□□□sigaction(SIGQUIT,□&q_act,□&o_act);
□□□sigaction(SIGHUP,□&h_act,□&o_act);
□□□return(c_stat);
```

}

First the return value of *pclose* indicates whether the process executed successfully. If If a prrocess spawns multiple child processes, it is equally important that there should be only a limited number of child processes that are not waited for. This still applies even if some of them havae already been terminated. The call to *wait* terminates the child process. The to *sigaction* ensures that the process will not be affected by interrupts etc. while in the wait state (see *sigaction*(2)).

The routine as written has the drawback that only one interprocess channel can be open at a time owing to the single shared variable *popen_pid*. There should really be an array with file descriptors as subscripts. The standard I/O library supplies a function similar to *popen*, but this function has slightly different arguments and a different return value (see *stdio(3S)*).

3.4 File and record locking

Mandatory and advisory file and record locking both are available on current releases of the Reliant UNIX system. The intent of this capability to is provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multiuser applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like *euug*, a European organization of UNIX system users from businesses and research bodies.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

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

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

3.4.1 Terminology

Before discussing how record locking should be used, let us first define a few terms.

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

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

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

Write (Exclusive) Locks
: These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking
: A form of record locking that does not interact with the I/O subsystem. Advisory locking is not enforced,

for example, by *creat(2)*, *open(2)*, *read(2)*, or *write(2)*. The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

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

3.4.2 File protection

There are access permissions for Reliant UNIX system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit of the data base accessing programs; see *chmod(1)* on .... 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 lokking is not used, is the *mail(1)* command. In that command only the particular user and the *mail* command can read and write in the unread mail files.

3.4.2.1 Opening a file for record locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. For our example we will open our file for both read and write access:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
int fd;          /* file descriptor*/
char *filename;
main(argc, argv)
int argc;
char *argv[];
{
      extern void exit(), perror();
      /* get data base file name from command line and open the
       * file for read and write access.
       */
      if (argc < 2) {
            (void) fprintf(stderr, "usage: %s filename\n", argv[0])
            exit(2);
      }
      filename = argv[1];
      fd = open(filename, O_RDWR);
```

```
□□□□□□if□(fd□<□0)□{
□□□□□□□□perror(filename);
□□□□□□□□□□□□exit(2);
□□□□□□□□□□□□}
□□□□□□.
□□□□□□.
□□□□□□.
```

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

Files mapped in the process address space cannot be locked: if a file has been mapped, any attempt to use file or record locking on the file fails. See *mmap(2)*.

3.4.2.2 Setting a file lock

There are several ways for us to set a lock on a file. In part, these methods depend on how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the *fcntl(2)* system call, the other using the *EUUG* standards compatible *lockf(3C)* library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the *fcntl* system call is as follows:

```
#include□<fcntl.h>
#define□MAX_TRY10
int□try;
struct□flock□lck;
try□=□0;
/*□set□up□the□record□locking□structure,□the□address□of□which
*□is□passed□to□the□fcntl□system□call.
*/
lck.l_type□=□F_WRLCK;□□□□□□□□□/*□setting□a□write□lock□*/
lck.l_whence□=□0;□□□□□□□□□□□□□/*□offset□l_start□from□beginning□of□file□*/
lck.l_start□=□0L;
lck.l_len□=□0L;□□□□□□□□□□□□□□□/*□until□the□end□of□the□file□address□space□*/
/*□Attempt□locking□MAX_TRY□times□before□giving□up.
*/
while□(fcntl(fd,□F_SETLK,□&lck)□<□0)□{
□□□□□□□□□□if□(errno□==□EAGAIN□||□errno□==□EACCES)□{
□□□□□□□□□□□□□□□□□□□□□□/*□there□might□be□other□errors□cases□in□which
□□□□□□□□□□□□□□□□□□□□□□*□you□might□try□again.
□□□□□□□□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□□□□□□□□if□(++try□<□MAX_TRY)□{
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□(void)□sleep(2);
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□continue;
□□□□□□□□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,"File□busy□try□again□later!\n");
□□□□□□□□□□□□□□□□□□□□□□return;
□□□□□□□□□□}
□□□□□□□□□□perror("fcntl");
□□□□□□□□□□exit(2);
□}
□□□□□□□□□□.
□□□□□□□□□□.
□□□□□□□□□□.
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked

- an error occurs
- it gives up trying because *MAX_TRY* has been exceeded

To perform the same task using the *lockf(3C)* function, the code is as follows:

```
#include <unistd.h>
#define MAX_TRY10
int try;
try = 0;
/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);
/* Attempt locking MAX_TRY times before giving up.
*/
while (lockf(fd, F_TLOCK, 0L) < 0) {
                if (errno == EAGAIN || errno == EACCES) {
                        /* there might be other errors cases in which
                         * you might try again.
                         */
                        if (++try < MAX_TRY) {
                                sleep(2);
                                continue;
                        }
                        (void) fprintf(stderr,"File busy try again later!\n");
                        return;
                }
                perror("lockf");
                exit(2);
}
.
.
.
```

It should be noted that the *lockf* example appears to be simpler, but the *fcntl(2)* example exhibits additional flexibility. Using the *fcntl(2)* method, it is possible to set the type and start of the lock request simply by setting a few structure variables. *lockf* merely sets write (exclusive) locks; an additional system call, *lseek*, is required to specify the start of the lock.

3.4.2.3 Setting and removing record locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, 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 if 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
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

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

record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the *euug lockf* function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```
struct□record□{
□□□□□.
□□□□□.□□□□□□□□/*□data□portion□of□record□*/
□□□□□.
□□□□□long□prev;□□□□□□□□/*□index□to□previous□record□in□the□list□*/
□□□□□long□next;□□□□□□□□/*□index□to□next□record□in□the□list□*/
};
/*□Lock□promotion□using□fcntl(2)
□*□When□this□routine□is□entered□it□is□assumed□that□there□are□read
□*□locks□on□"here"□and□"next".
□*□If□write□locks□on□"here"□and□"next"□are□obtained:
□*□□□□Set□a□write□lock□on□"this".
□*□□□□Return□index□to□"this"□record.
□*□If□any□write□lock□is□not□obtained:
□*□□□□Restore□read□locks□on□"here"□and□"next".
□*□□□□Remove□all□other□locks.
□*□□□□Return□a□-1.
□*/
long□set3lock□(this,□here,□next)
long□this,□here,□next;
{
□□□□□□struct□flock□lck;
□□□□□□lck.l_type□=□F_WRLCK;□□□/*□setting□a□write□lock□*/
□□□□□□lck.l_whence□=□0;□□□□□□□/*□offset□l_start□from□beginning□of□file□*/
□□□□□□lck.l_start□=□here;
□□□□□□lck.l_len□=□sizeof(struct□record);
□□□□□□/*□promote□lock□on□"here"□to□write□lock□*/
□□□□□□if□(fcntl(fd,□F_SETLKW,□&lck)□<□0)□{
□□□□□□□□□□□□□□return□(-1);
□□□□□□}
□□□□□□/*□lock□"this"□with□write□lock□*/
□□□□□□lck.l_start□=□this;
□□□□□□if□(fcntl(fd,□F_SETLKW,□&lck)□<□0)□{
□□□□□□□□□□□□/*□Lock□on□"this"□failed;
□□□□□□□□□□□□□□*□demote□lock□on□"here"□to□read□lock.
□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□lck.l_type□=□F_RDLCK;
□□□□□□□□□□□□□□lck.l_start□=□here;
□□□□□□□□□□□□□□(void)□fcntl(fd,□F_SETLKW,□&lck);
□□□□□□□□□□□□□□return□(-1)
□□□□□□}
□□□□/*□promote□lock□on□"next"□to□write□lock□*/
□□□□□lck.l_start□=□next;
□□□□□if□(fcntl(fd,□F_SETLKW,□&lck)□<□0)□{
□□□□□□□□□□□□□□/*□Lock□on□"next"□failed;
□□□□□□□□□□□□□□*□demote□lock□on□"here"□to□read□lock,
□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□lck.l_type□=□F_RDLCK;
□□□□□□□□□□□□□□lck.l_start□=□here;
□□□□□□□□□□□□□□(void)□fcntl(fd,□F_SETLK,□&lck);
□□□□□□□□□□□□□□/*□and□remove□lock□on□"this".
□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□lck.l_type□=□F_UNLCK;
□□□□□□□□□□□□□□lck.l_start□=□this;
□□□□□□□□□□□□□□(void)□fcntl(fd,□F_SETLK,□&lck);
```

```
□□□□□□□□□□□□□□return□(-1);
□□□□/*□cannot□set□lock,□try□again□or□quit□*/
□□□□}
return□(this);
}
```

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

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

Lock promotion using lockf(3)

```
/*□Lock□promotion□using□lockf(3)
*□When□this□routine□is□entered□it□is□assumed□that□there□are
*□no□locks□on□"here"□and□"next".
*□If□locks□are□obtained:
*□□□□Set□a□lock□on□"this".
*□□□□Return□index□to□"this"□record.
*□If□any□lock□is□not□obtained:
*□□□□Remove□all□other□locks.
*□□□□Return□a□-1.
*/
#include□<unistd.h>
long
set3lock□(this,□here,□next)
long□this,□here,□next;
{
□□□□□/*□lock□"here"□*/
□□□□□(void)□lseek(fd,□here,□0);
□□□□□if□(lockf(fd,□F_LOCK,□sizeof(struct□record))□<□0)□{
□□□□□return□(-1);
□□□□□}
□□□□□/*□lock□"this"□*/
□□□□□(void)□lseek(fd,□this,□0);
□□□□□if□(lockf(fd,□F_LOCK,□sizeof(struct□record))□<□0)□{
□□□□□□□□□□□/*□Lock□on□"this"□failed.
□□□□□□□□□□□*□Clear□lock□on□"here".
□□□□□□□□□□□*/
□□□□□□□□□□(void)□lseek(fd,□here,□0);
□□□□□□□□□□(void)□lockf(fd,□F_ULOCK,□sizeof(struct□record));
□□□□□□□□□□return□(-1);
□□□□□}
□□□□□/*□lock□"next"□*/
□□□□□(void)□lseek(fd,□next,□0);
□□□□□if□(lockf(fd,□F_LOCK,□sizeof(struct□record))□<□0)
□□□□□{
□□□□□□□□□□/*□Lock□on□"next"□failed.□/
□□□□□□□□□□*□Clear□lock□on□"here",
□□□□□□□□□□*/
□□□□□□□□□□(void)□lseek(fd,□here,□0);
□□□□□□□□□□(void)□lockf(fd,□F_ULOCK,□sizeof(struct□record));
□□□□□□□□□□/*□and□remove□lock□on□"this".
□□□□□□□□□□*/
```

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

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

3.4.2.4 Getting lock information

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

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

Printing segments locked by other processes

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

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

The *lockf* function with the *F_TEST* command can also be used to test if there is a process blocking a lock.

This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using *lockf* to test for a lock on a file follows:

```
/*□find□a□blocked□record□*/
/*□seek□to□beginning□of□file□□*/
(void)□lseek(fd,□0,□0L);
/*□set□the□size□of□the□test□region□to□zero□(0)
□*□to□test□until□the□end□of□the□file□address□space.
□*/
if□(lockf(fd,□F_TEST,□0L)□<□0)□{
□□□□□switch□(errno)□{
□□□□□□□□□□□□case□EACCES:
□□□□□□□□□□□□case□EAGAIN:
□□□□□□□□□□□□(void)□printf("another□process□locking□file\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 *fork*s, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by *l_start*, when using a *l_whence* value of *1*. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the *lockf(3)* function call as well and is a result of the *euug* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring.

Another solution is to use the *fcntl* system call with a *l_whence* value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

3.4.2.5 Deadlock handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the *euug* standard *lockf* call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set *errno* to the deadlock error number. If a process wishes to avoid the use of the system's deadlock detection it should set its locks using *F_GETLK* instead of *F_GETLKW*.

3.4.3 Selecting advisory or mandatory locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section (see Section "Caveat emptor mandatory locking"). Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made by the permissions on the file; see *chmod(2)*. For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

```
#include□<sys/types.h>
```

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

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

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

chmod +l file

The *ls(1)* command shows this setting when you ask for the long listing format:

ls -l file

causes the following to be printed:

-rw---l--- 1 user group size mod_time file

3.4.3.1 Caveat emptor — mandatory locking

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

3.4.3.2 Record locking and future releases of the Reliant UNIX system

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

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

3.5 Display and keyboard I/O

This section describes how to manage text and graphics output in Reliant UNIX 5.43. Most of the content is also applicable to SINIX V5.4x.

This section is intended primarily for programmers concerned with developing low-level graphics applications. The aim of this section is to enable you to get at the internal hardware controls necessary to make effective use of the video adapters; it is not concerned with higher level graphics concerns such as shading or rotating images.

While it would be useful to have some familiarity with the PC AT architecture and understand the basics of video programming, it is not necessary. You should, however, be know-ledgeable of Reliant UNIX 5.43 for the C programming language, since all examples are written in C.

The section is divided into several distinct subsections. The first subsections present both a conceptual and technical overview of developing programs (see ...) that make full use of the text and graphics facilities. The next subsections present the same material from a more specific technical viewpoint and provide annotated programming examples. They describe the specific requirements for text mode programming (see ...), the use of video memory (see ...), and graphics mode programming (see ...), including specific requirements for programming the video control registers (see ...).

One of the significant features of the video interface is its Virtual Terminal Capability. This capability extends the notion of windowing to the next level and allows for controlling several independent windowing applications. After presenting the initial view of graphics programming, there is an extensive subsection that describes the Virtual Terminal Capability and provides extensive programming examples of how to make effective use of that capability (refer to Section "Using Virtual Terminals").

The last subsections present various miscellaneous features, including setting borders (see ...), keyboard operations (see ...), sound control (see ...), font operations (see ...) and programming the mouse see ...).

This section is not a tutorial; rather it summarizes and organizes the most common and frequently used technical information and provides extensive examples demonstrating the use of the video capabilities of the video interface.

The primary source of information for effectively programming the video interface is: the *display(7)* and *keyboard(7)* man pages, either on-line or in the corresponding sections of [7].

The *display(7)* manual pages describe both text and graphics control of the display. For text output, the manual pages present the effect of different output character sequences on the display. For graphic control, the manual pages describe the function call details of the *ioctl* system routine (see *ioctl(2)* in [9]) that provides the graphics interface to the display and to the *keyboard/display* (kd) driver known as the *kd* driver. Both the *display(7)* and *keyboard(7)* man pages include necessary information for using and controlling Virtual Terminals.

You should refer to the manuals that come with the video board you are using for information on the hardware layout, the differences among different video boards, the memory maps, and video registers.

3.5.1 Conceptual overview of video display programming

3.5.1.1 Video adapter boards

The video display is physically controlled by hardware boards called video adapters. The video adapters essentially determine the resolution and color possibilities of the display image in both text and graphics modes. Adapters will differ in the number of colors that are possible and the maximum possible screen resolution.

The first step in video display programming, then, is to determine the specific adapter board (sometimes called an adapter card) that is being used. It is possible to develop programs for the lowest level board, in

which case this step is not necessary. In most cases, however, you would want to make use of the maximum capabilities of the board, to have the highest possible screen resolution, in which case it is essential that you know the board you are using.

There are many widely used standard video adapters. These include:

- The MDA (Monochrome Display Adapter) is a basic 2 color display, i.e. black and one other. It has a resolution of 720 pixels across by 350 pixels down.
- The CGA (Color Graphics Adapter) provides 4 colors, with a 320 x 200 resolution, or 2 colors, with a 640 x 200 resolution.
- The EGA (Enhanced Graphics Adapter) provides 16 colors and 640 x 350 pixel resolution.
- The VGA (Video Graphics Adapter) provides 16 colors, with 640 x 480 pixel resolution and 256 colors at 320 x 200 pixel resolution.

You may also use a Hercules monochrome graphics adapter and other commonly available graphics adapters. Each controller/adapter has different capabilities, registers, and memory mapping. In Section "Graphics mode", the characteristics of each of the graphics modes available are summarized.

Determining the adapter

Accessing the VDC requires two programming steps:

- Opening the Device
- Getting the Adapter Information

Like all devices in Reliant UNIX 5.43, the adapter is opened as if it were a file. A specific statement to use is:

open("/dev/video", O_RDWR);

The expectation in opening */dev/video* is that the controlling tty is a Virtual Terminal. The *open* will fail if the device is connected via a serial terminal port. If the device cannot be opened, the error return is a negative number. The *ioctl(2)* system call is used to gather information, set modes, issue miscellaneous commands, etc. Because of the historical evolution of Reliant UNIX 5.43, there are several different versions of the *ioctl(2)* system call that either can be used or have to be used, depending on the specific information required. The techniques used in the examples are recommended. They provide a cleaner interface to the driver code and are essential for the effective use of Virtual Terminals. These calls and variations are:

- ioctl(fd, KIOCINFO, 0);

  This call is used to determine if the file description given by *fd* is for a device that can be controlled by the *kd* driver. If it is, the return will be (*'k'<<8 | 'd'*).

- ioctl(disp, KDVDCTYPE, &disp_info);

  The argument, *&disp_info*, is the address of a structure that is filled by the called routine. For example, it may be defined by:

  struct□kd_vdctype□□□□disp_info;

  On return from the *ioctl* routine, the structure will contain the controller type and the display type. The mnemonics used are defined in */usr/include/sys/kd.h*.

  Return values for the controller field are:

Return	Adapter type
KD_MONO	IBM monochrome display adapter.
KD_HERCULES	Hercules monochrome adapter
KD_CGA	IBM colorgraphics adapter
KD_EGA	IBM enhanced graphics adapter
KD_VGA	IBM video graphics adapter
KD_VDC400	AT&T VDC 400 adapter

KD_VDC750	AT&T VDC 750 adapter
KD_VDC600	AT&T VDC 600 adapter

Table 20: Return values for the controller field

and the standard returns for the monitor type are:

Return	Monitor type
KD_UNKNOWN	Unknown monitor type
KD_STAND_M	Standard monochrome monitor
KD_STAND_C	Standard color monitor
KD_MULTI_M	Multi-mode monochrome monitor
KD_MULTI_C	Multi-mode color monitor

Table 21: Standard returns for the monitor type

If the *ioctl(2)* system call function returns a value of -1, a data transfer error occurred.

The following code fragment illustrates how to find the adapter type:

```
{
□□□□□...
□□□□□struct□kd_vdctype□vdcinfo;
□□□□□...
□□□□□/*issue□the□ioctl□to□get□the□VDC□type□□*/
□□□□□if□(ioctl(disp,□KDVDCTYPE,□&vdcinfo)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"KDVDCTYPE□failed");
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□LOOK□AT□THE□ADAPTER□TYPE□□□□□□*/
□□□□□switch□(vdcinfo.cntlr)□{□□/*□switch□on□the□adapter□type□*/
□□□□□case□KD_EGA:
□□□□□□□□□□□□printf("EGA□Compatible□Adapter□Unit□\n");
□□□□□□□□□□□□break;
□□□□□case□KD_VGA:
□□□□□□□□□□□□printf("VGA□Compatible□Adapter□Unit□\n");
□□□□□□□□□□□□break;
□□□□□default:
□□□□□□□□□□□□printf("This□application□will□only□run□on□systems\n");
□□□□□□□□□□□□printf("configured□with□EGA□or□VGA□compatible\□controllers\n");
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□LOOK□AT□THE□MONITOR□TYPE□□□□□□*/
□□□□□switch□(vdcinfo.disply)□{□□/*□switch□on□monitor□type□□*/
□□□□□case□KD_STAND_M:
□□□□□case□KD_MULTI_M:
□□□□□□□□□□□□printf("Warning:□This□application□requires□a□color\n");
□□□□□□□□□□□□printf("monitor□for□some□of□the□options.\n");
□□□□□□□□□□□□break;
□□□□□case□KD_STAND_C:
□□□□□case□KD_MULTI_C:
□□□□□□□□□□□□printf("Color□Monitor□\n");
□□□□□□□□□□□□break;
□□□□□default:
□□□□□□□□□□□□printf("Warning:□Unknown□monitor□type.\n");
□□□□□□□□□□□□break;
□□□□□}
□□□□□...
}
```

Getting and setting the mode

The mnemonics for setting the adapter mode are of the form SW_type, where 'type' indicates the combination

of adapter and mode. For example, SW_C80x25 is used to set the mode to CGA text, 80 x 25 while SW_VDC640x400V selects the 640 x 400 graphics mode.

The mode is set by calling the routine:

ioctl(disp, SW_type, 0);

A negative return indicates an error.

The CONS_GET routine is used to determine the current adapter mode. It returns the mode using the mnemonic DM_type, where 'type' is as above. The mode is obtained by calling:

ioctl(disp, CONS_GET, 0);

The return is the current mode setting, or a value less than zero if there were an error.

The relationship between the mode as gotten and the mode to be set is:

SW_type = DM_type | MODESWITCH;

That is, the value used to set the mode can be obtained by or-ing a previously gotten and saved mode value with MODESWITCH.

A call to *ioctl(disp, KDSETMODE, KD_TEXT);* sets the adapter from whatever its current mode is to the default text mode and also clears the screen.

One reason for having to save and restore adapter modes is that a video application may be invoked within other video applications.

The example below illustrates the operation just described. The code fragment opens the device in one mode; gets and saves that mode; switches modes and then restores the original mode. Prudent programming will also catch all catchable signals in order to restore the initial settings.

Restoring the adapter modes

```
#□include□<stdio.h>
#□include□<fcntl.h>
#□include□<sys/kd.h>
#□include□<errno.h>
main()
{
□□□□□int□□□□□disp,□save_mode;
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□<□0)□{
□□□□□□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno□%d\n",
□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□set□the□mode□to□wide□text□and□print□something□out□*/
□□□□□if□(ioctl(disp,□SW_C40x25,□0)□<□0)□{
□□□□□□□□□□□□□□□□□fprintf(stderr,□"SW_C40x25□ioctl□failed\n");
□□□□□□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□printf("Testing□SW_C40x25□ioctl\n");
□□□□□sleep(2);
□□□□□/*□Save□that□mode□*/
□□□□□if□((save_mode□=□ioctl(disp,□CONS_GET,□0))□<□0)□{
□□□□□□□□□□□□□□□□□fprintf(stderr,□"CONS_GET□failed");
□□□□□□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Clear□the□screen□and□reset□the□display
□□□□□□□□back□to□the□default□text□mode:□*/
□□□□□if□(ioctl(disp,□KDSETMODE,□KD_TEXT)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"Unable□to□reset□display,□error:□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□□/*□Show□normal□text□*/
□□□□□printf("Now□in□normal□narrow□text□mode□");
□□□□□sleep(2);
□□□□□/*□Return□to□wide□text□*/
```

```
□□□□□if□(ioctl(disp,□MODESWITCH□|□save_mode,□0)□<□0)□{
□□□□□□□□□□□□□□□□□□fprintf(stderr,□"Unable□to□reset□display,□mode:□%x\n",
□□□□□□□□□□□□□□□□□□save_mode);
□□□□□□□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□exit(0);
}
```

3.5.1.2 Memory

All access to physical hardware is regulated by the Reliant UNIX system kernel and device drivers. In the case of the console video graphics controller, the *kd* display driver is charged with this responsibility.

To provide a reasonable level of protection, most of the common controller operations are supported via the *ioctl(2)* system call interface. Examples of such operations include: video mode selects, controller status, I/O operations to the controller, and get/set screen attributes. In addition to these operations for interacting with the controller, one additional important feature is provided; the ability to map video memory into your address space.

From a programmer's viewpoint, the adapter card can be considered to be directly linked to memory locations within the computer. Changing the contents of the memory immediately changes the content of the display. An essential element of programming the video display, then, is understanding exactly how the memory corresponds to the output. The specific way in which memory must be laid out is a function of the adapter card and the mode with which that adapter is being used. The specific layout and mapping of memory are described in subsequent sections.

3.5.1.3 Registers

The actual operation of the adapter is controlled by certain registers on the adapter.

In Reliant UNIX 5.43, the *kd* driver does all the basic video register setup at the time you select the graphics mode. The driver has been designed to provide good performance while maintaining a safe user level interface. The driver does not fully protect you against programming mistakes. On the contrary, you are specifically allowed to read and write the video hardware registers directly. Some of these registers should not be directly written unless you are fully aware of the consequences and have taken steps to ensure the correctness of the operation.

3.5.1.4 Text mode and graphics mode

A critical aspect to video display programming is the distinction between text mode and graphics mode. The two modes are mutually exclusive; an adapter can only be in one of those two modes, not both. Thus, a programmer can produce either a graphics display or a text display, but not both at the same time.

Text mode

Under text mode, you can specify the text format and character set; you can display foreground and background colors; you can control cursor movement and can even control character mapping. In text mode, the display resolution is considered in terms of characters; for example an 80 x 25 resolution screen can hold 80 characters across the width of the screen and 25 lines down the screen. Text mode is the default mode.

There are several different character sets that you can choose from. Some of these sets provide 'graphic' characters, for example arrows, lines, corners, etc. Because text mode is both faster and easier to use than graphics mode, you should consider whether it makes sense to do your application entirely in text mode.

There are three kinds of output possible in text mode. These can be loosely characterized as "'standard character output", "non-standard character output" and "Escape code sequences".

- In standard output you "see what you say"; that is, the statement in your program says *printf("XYZ")*; and the characters *XYZ* appear on the display.
- Non-standard output describes what happens when you output the non-printing characters such as LF, FF, CR, BEL, etc. The *display(7)* manual pages describe the action that occurs when each of these characters is output. For example, outputting FF will clear the screen and put the cursor at line 1, column 1.
- Escape sequences allow you to control the cursor movement, perform screen editing, assign values to function keys, and specify attributes of each character. Escape sequences also let you manage screen

input as well as output. For example, you can program the effect of function keys on the display, lock and unlock the keyboard, etc. The escape sequences are also described in detail in the *display(7)* manual page. Several of the escape sequences are used as examples in the next subsection (see Section "Escape sequences").

Graphics mode

In graphics mode, the display is viewed as consisting of addressable points called picture elements (or pixels). The combination of the adapter board and the display mode establish a screen resolution and color capability. You can address each pixel of the display and specify the color of each display pixel.

In both text and graphics mode, you can read and set video hardware registers, and create Virtual Terminals.

The adapters respond to additional commands that allow you to read and set the keyboard LEDs, generate sounds and tones, and perform other miscellaneous functions.

3.5.1.5 Steps in programming

Before you start

You will need to become familiar with the *ioctl(2)* system call and the header file, */usr/include/sys/kd.h*. The header file defines the display vocabulary (i.e. the complete set of *#defines*) that you need in order to use the *ioctl(2)* system call. It also contains the definitions of those structures that allow you to read and write the control registers. The other header files that you will require are */usr/include/sys/types.h* and */usr/include/sys/at_ansi.h*.

Aside from normal design considerations, the two most important decisions you must make before you start are:

- Will I be in text mode or graphics mode?
- What board should I program for; more specifically, what resolution should I program for?

Text mode

The programmatic steps are:

- Create a set of *#defines* for the escape sequences you will use. This is not a programming requirement, but it is good programming practice and results in more readable and maintainable code.
- Open the *kd* driver.
- Determine which adapter board is attached.
- Establish the text mode, if you do not intend to use the default mode.
- Clear the screen (output the sequence ESCc, i.e., *printf("\033c");*).
- Output the text.

  There are two general ways to output text. The first is by using straightforward output functions, such as *printf*. In this mode, the escape sequences are used to control foreground and background color as well as other text attributes, such as blink or underscore.

  The second is by writing directly to the video memory. You can actually work with several screenloads at once. The amount of memory available to the video adapter is a multiple of 16K bytes, depending on which adapter is used. The amount used for a text screen is either 2K or 4K, depending on the mode selected. This means that there can be at least 4 to 8 text screens stored simultaneously. You control which of those screens is currently displayed and you can switch the displays instantaneously (at least from the viewpoint of the end-user).

Graphics mode

In order to be effective in using graphics mode, you must develop a basic library of primitive routines. All graphics programming is a function of writing to memory and mapping the memory to the display screen.

The getting started steps are the same for graphics mode as for text mode and consist of:

- Open the *kd* driver.
- Determine the adapter board.

- Establish the Graphic Mode (a combination of number of colors and resolution).
- Clear the screen.

  This consists of writing the appropriate foreground/background pixel value into all of video memory. The actual value written depends on the background attribute.
- Get to work.

Further on in this section there is a complete graphics programming example (see Section "Graphics mode").

3.5.2 Programming in text mode

3.5.2.1 How text is stored

In text mode, each character is stored in memory as two data bytes. The first one is for the character itself and the second one is for its attributes. A character's attributes specify the foreground and background colors, as well as whether that character is to be highlighted, underlined, or blinking. These characteristics differ somewhat, depending on the video mode and the adapter, and also on whether the display screen is monochrome or color.

Bits 0 - 3 affect the Foreground Color; bits 4 - 7 affect the Background Color.

The adapters provide alternate character sets. We recommend that you run a simple program that exercises the different character set options. The following code fragment is illustrative:

```
{
□□□□□NOTE:□Escape□(ESC)□is□octal□33
□□□□□/*□Clear□the□screen□and□select□1st□alt□char□set.□*/
□□□□□printf("\033c\033[11m")
□□□□□for□(i□=□0;□i□<□255;□i++)
□□□□□□□□□□□printf("%c",(char)i);
□□□□□/*□Select□2nd□alt□char□set.□*/
□□□□□printf("\n\n\033[12m");
□□□□□for□(i□=□0;□i□<□255;□i++)
□□□□□□□□□□□printf("%c",(char)i);
□□□□□/*□Restore□the□primary□character□set.□*/
□□□□□printf("\033[10m");
}
```

Many video display controllers allow additional character sets or text fonts to be created locally by an application programmer. For example, some VDCs allow 16 such fonts and others may allow 8 such fonts. Each character set is stored in a fixed section of memory and is selected by setting the Expanded Character Select Register. Register programming will be described in Section "Accessing video controller registers".

3.5.2.2 Selecting the text mode

Each of the adapters allows several possible text modes.

- One approach to selecting a mode is to choose the highest resolution mode available on the adapter.
- If compatibility is an issue, select the mode that will work on the lowest common adapter that the application will run on.

Assuming an 80 column width, the difference between text modes on the same adapter is that the actual characters can be displayed with different resolutions. For example, a VDC that can operate compatibly with CGA, EGA or VGA applications can display characters in 3 different 80 x 25 character modes.

- VGA mode 2 displays characters in a 9 x 16 box (144 pixels),
- EGA mode 2 displays characters in an 8 x 14 box (112 pixels).
- CGA mode 2 displays characters in an 8 x 8 box (64 pixels).

The VGA and EGA's higher resolutions produce characters that are crisper and easier to read.

In Section "Text and graphics mode ioctls", all text and graphics modes are summarized. It lists the defined mnemonic used in the *ioctl(2)* system call and the adapter type it can work on.

3.5.2.3 Escape sequences

Escape sequences are used for three purposes:

- To change the characteristics of the text to be displayed.

  These characteristics can also be altered by directly addressing the video memory. This aspect of video programming is discussed in one of the next subsections.
- To program some of the Video Registers and affect the overall display.

  For example, Escape sequences can modify those registers that select the character set, move the cursor, erase all or part of the screen, etc.
- To issue some miscellaneous commands.

  For example, you can issue an Escape sequence (*ESC[1k*) that produces a "click" each time the user presses a key. The sequence *ESC[0k* disables that feature.

The Escape sequences can be issued at the command level; in particular, they can be embedded in shell scripts that are integral to an application system. For example, including the following within a user's *.profile* displays the primary prompt as the machine name in cyan:

```
UNAME='uname'
PS1='echo "\\033[36m$UNAME: \\033[37m"'
```

The Escape sequences adhere to the ANSI X3.64 standard for ASCII terminals, with a few extensions for color and enabling the key-click feature. The complete list of escape commands is included in the *display(7)* documentation.

The following C program displays the characters of the alphabet in different foreground and background colors:

```
main ()
{
     int i, j;
     char *string = "abcdefghijklmnopqrstuvwxyz"
     for (i = 30; i < 38; i++)
          for (j = 40; j < 48; j++)
               printf("\033[%d;%dm%s\n", i, j, string);
     /* Restore the default white on black display colors: */
     printf("\033[0m\n");
}
```

3.5.2.4 Example of text mode programming

The following program opens the video adapter and displays the printable character set under whatever text modes are valid on that adapter. Not all monitors will support all modes.

Displaying the printable character set

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/at_ansi.h>
#include <sys/kd.h>
#include <errno.h>
struct {
     unchar       m_name[20];   /* ioctl name */
     int          m_value;      /* ioctl value */
} modes[] = {
     { "SW_B40x25",          SW_B40x25 },
     { "SW_C40x25",          SW_C40x25 },
     { "SW_B80x25",          SW_B80x25 },
     { "SW_C80x25",          SW_C80x25 },
     { "SW_ENHB40x25",       SW_ENHB40x25 },
     { "SW_ENHC40x25",       SW_ENHC40x25 },
     { "SW_ENHB80x25",       SW_ENHB80x25 },
```

```
□□□□□{□"SW_ENHC80x25",□□□□□SW_ENHC80x25□□□□},
□□□□□{□"SW_EGAMONO80x25",□□SW_EGAMONO80x25□},
□□□□□{□"SW_ENHB80x43",□□□□□SW_ENHB80x43□□□□},
□□□□□{□"SW_ENHC80x43",□□□□□SW_ENHC80x43□□□□},
□□□□□{□"SW_VGAC40x25",□□□□□SW_VGAC40x25□□□□},
□□□□□{□"SW_VGAC80x25",□□□□□SW_VGAC80x25□□□□},
□□□□□{□"SW_VGAMONO80x25",□□SW_VGAMONO80x25□}
};
main()
{
□□□□□int□□□i,□j;
□□□□□int□□□disp;
□□□□□/*□Standard□form□for□opening□the□driver.□□If□the□open□is
□□□□□□□□successful,□the□controlling□tty□is,□in□fact,□a□valid
□□□□□□□□display□device.□□*/
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□error:□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□/*Try□to□set□every□possible□text□mode.□□Only□the□valid□modes□will□work.□*/
□□□□□for□(i□=□0;□i□<□14;□i++)□{
□□□□□□□□□□□□if□(ioctl(disp,□modes[i].m_value,□0)□<□0)□{
□□□□□□□□□□□□□□□□□□printf("\033c%s□not□supported.",□modes[i].m_name);
□□□□□□□□□□□□□□□□□□fflush(stdout);
□□□□□□□□□□□□}□else□{
□□□□□□□□□□□□□□□□□□/*□For□every□valid□mode,□output□every□printable□character.□*/
□□□□□□□□□□□□□□□□□□printf("\033c%s\n",□modes[i].m_name);
□□□□□□□□□□□□□□□□□□for□(j□=□0x21;□j□<□0x7f;□j++)□{
□□□□□□□□□□□□□□□□□□□□□□printf("%c",□j);
□□□□□□□□□□□□□□□□□□□□□□if□(!((j□-□0x20)□%□24))
□□□□□□□□□□□□□□□□□□□□□□□□□□□□printf("\n");
□□□□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□□□□fflush(stdout);
□□□□□□□□□□□□}
□□□□□□□□□□□□sleep(1);
□□□□□□□□□□□□sleep(1);
□□□□□}
□□□□□/*Clear□the□screen□and□reset□the□display□back□to□the□default□text□mode*/
□□□□□if□(ioctl(disp,□KDSETMODE,□KD_TEXT)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"Unable□to□reset□display,□error:□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
}
```

3.5.2.5 Text programming memory management

The ability to access video memory is a necessary feature for developing high performance graphics applications. The information that appears on the screen is directly correlated to a block of memory. Writing to memory directly is faster than going through intermediate routines. And changing video memory produces instant change on the screen.

In text mode you can store several screen loads in video memory at the same time. This gives you the ability to write to one screen without disturbing what the end user sees and then switching to a new, full screen's worth of data. There is a tradeoff in using this particular technique. Its downside is that using memory switching bypasses the *kd* driver control which means that switching Virtual Terminals may not be effective. This means you would have to seize the screen during the operation (by entering VT_PROCESS mode, which is described later).

Memory layout

The specific layout of video memory is a function of the type of display and the video mode that is being used. The combination of the video mode and the display controller type determine both the amount of memory that needs to be mapped as well as where within video memory a particular screen's display data exists.

The table below describes the various video text modes and identifies the adapters that support each mode. The list is not exhaustive in that EGA and VGA adapters do support variations of the CGA modes. As an example, the mode 0 text mode for a VGA adapter has the ability to support the 40x25 character set in higher resolution (320 x 350 and 320 x 400) as opposed to the CGA's 320 x 200 pixel resolution.

Mode	Type	Resolution	Colors	Address	Type
0	Text	40 x 25	16 grey	0XB8000	CGA
1	Text	40 x 25	16f, 8b	0XB8000	CGA
2	Text	80 x 25	16 grey	0XB8000	CGA
3	Text	80 x 25	16f, 8b	0XB8000	CGA
7	Text	80 x 25	b/w	0XB0000	MDA

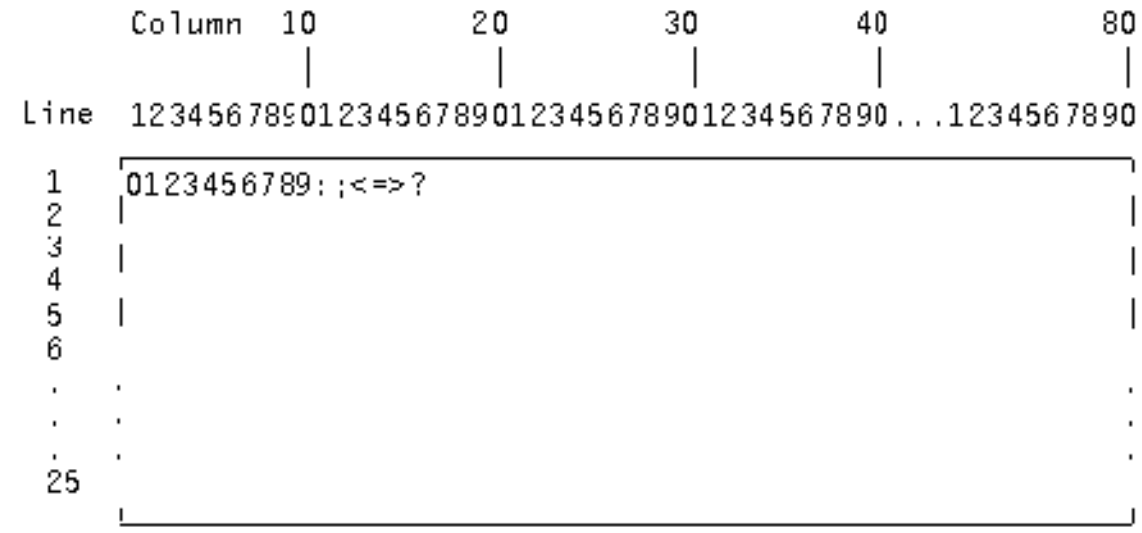
Table 22: Video text modes for CGA/EGA/VGA

Addressing of video memory in text mode is identical for all standard video systems (CGA, EGA, VGA). Two bytes of data are required for each character represented on the display screen. The first byte contains the ASCII character to be displayed and the second byte contains the attributes of that character.

For example, the memory required to display a single screen's data for a CGA 80 x 25 color adapter (mode 3) would be 25 rows * 80 lines * 2 bytes or 4000 bytes.

The attribute byte indicates the foreground/background colors, intensity, and underline. Consider the following memory dump for a single page:

□□displayed
□□□□□□□□□□□□□0□□1□□2□□3□□4□□5□□6□□7□□8□□9□□A□□B□□C□□D□□E□□F□□□□□data
□0xB8000□□□30□07□31□07□32□07□33□07□34□07□35□07□36□07□37□07□□01234567
□0xB8010□□□38□07□39□07□3a□07□3b□07□3c□07□3d□07□3e□07□3f□07□□89:;<=>?
□0xB8020□□□20□07□20□07□20□07□20□07□20□07□20□07□20□07□20□07□20□07
□0xB8F90□□□20□07□20□07□20□07□20□07□20□07□20□07□20□07□20□07□20□07
Contents□of□memory



The even-numbered bytes hold the character representation and the odd-numbered bytes hold the attributes. The attribute bytes (in this case the '07's) are not displayed; rather these are used by the controller to determine how to display the data. In this case the attribute being called for is simply white on a black background. The layout of the attribute byte is a function of the adapter. In this case, mode 3, the layout is:

Bit	Attribute

7	Blinking foreground component
6	Red background component
5	Green background component
4	Blue background component
3	Intensity
2	Red foreground component
1	Green foreground component
0	Blue foreground component

Table 23: Color attribute (mode 3)

Since (in this case) only 4000 characters are needed to represent an entire screen within memory, then the video memory within a CGA compatible controller is not being fully used. The remaining memory however, need not be wasted. It is possible to make use of the remaining video buffer for storing more than one screen's data at a time. This feature allows a programmer to switch what is currently displayed on the screen quickly simply by redirecting where the controller begins addressing. Writing an entire screen of data into video memory is much slower than redirecting the controller's address register.

If you are doing this, you are bypassing the *kd* driver control. Because of this, switching Virtual Terminals may not be effective. To ensure that it is you need to enter VT_PROCESS mode (described later in Section "Using Virtual Terminals").

Because the CGA controller supports 16K RAM (16384 bytes), there is the possibility of storing 4 screens of data in video mode 3 (8 in video modes 0 and 1). With a screen using 4000 bytes however, four screens of memory will consume only 16000 bytes of the 16384 available. Each screen of data is aligned on a 4K (4096) byte boundary and thus there are 96 bytes at the end of each screen of video memory that is not used. See the following diagram:

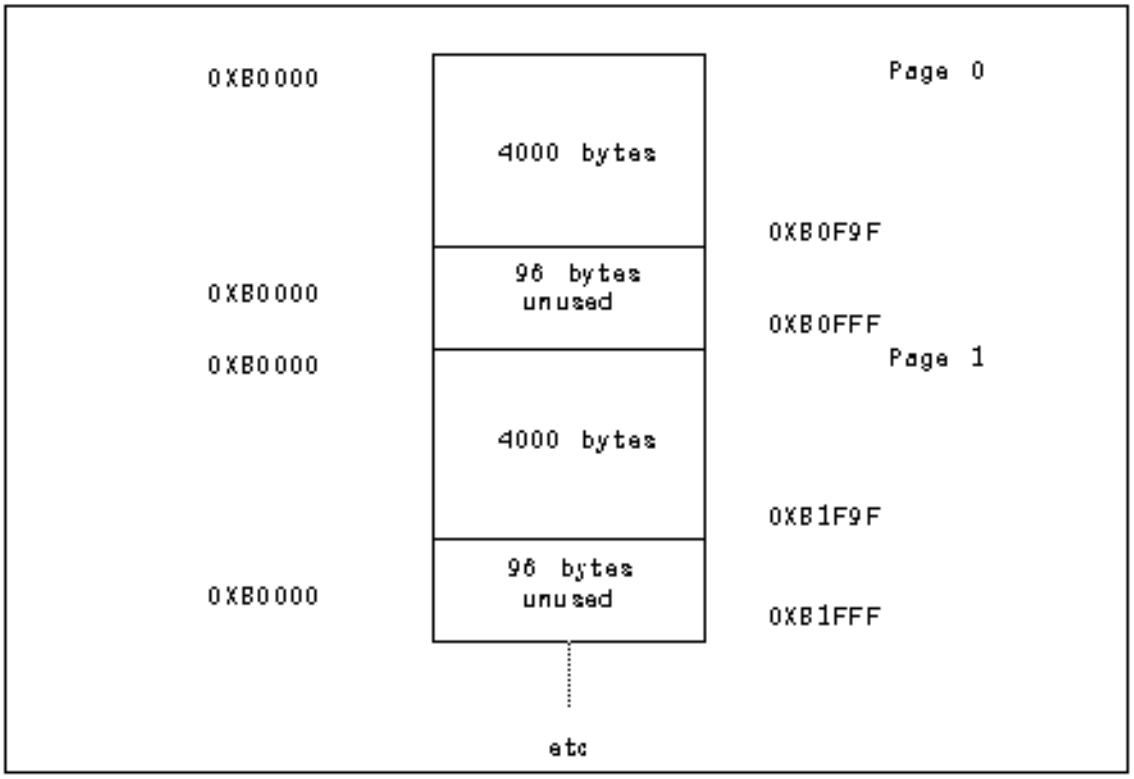


Figure 8: Screen alignment

The following formula provides one method of directly associating a byte position on the screen with an address in video memory:

video□buffer□address□=□video_memory_start□+□2□*□(80□*□row□+ column)

The video_memory_start is the start of the video buffer for the video mode we are using. The factor of two is necessary because of the attribute byte associated with each screen location. As an example, if we are using video mode 3 and wish to access the data and attribute for screen location row = 3, column = 30, then the video buffer address would be

0xB0000 + 2 * (80 * 3 + 30) = 0xB021C

B021C would be the address for the data byte and *B021D* would correspond to the attribute byte for location 3, 30 in page 0. If on the other hand we wished to address row 3, column 30 in page 1, *0x1000* would be added to this value producing *0xB121C*. The same types of calculations are applicable to other video text modes. The above calculations are based on the assumption that the upper left corner of the screen i.e the origin is (0, 0) and not (1, 1).

High speed interactions with the end-user can be accomplished:

- Read the current cursor position by reading the high/low cursor registers (refer to section entitled Register programming example).
- Read the data at the cursor position by reading memory.
- Write data to the cursor position by writing memory.
- Use Escape Sequences to reposition the cursor.

These four capabilities allow you to develop completely interactive full screen sessions.

3.5.3 Programming access to video memory

Gaining access to video memory is identical in both text and graphics mode. It is essentially a three-step process:

1. Open the *kd* driver. The *kd* driver is opened as the file */dev/video*.
2. Get the physical address of the display in memory.
3. Map the display memory into an area of your program's address space.

Getting the physical address

Use the KDDISPTYPE *ioctl* to retrieve the display memory start address. The way to use this *ioctl(2)* system call is to define a structure that the driver will fill. In using KDDISPTYPE, the argument passed in the call is a pointer to the structure. The structure used by KDDISPTYPE is:

```
struct□kd_disparam□{
□□□□long□type;□□□□□□□□□□□□□□□□□□□□/*□display□type□*/
□□□□char□*physaddr;□□□□□□□□□□□□□□/*□display□memory□address□*/
□□□□ushort□ioaddr[MKDIOADDR];□□□/*□valid□I/O□addresses□*/
};
```

The structure element *type* becomes the type of display (CGA, EGA, VGA, ...).

While this call works on all adapters and modes, not all display types are returned. The return types are limited to KD_MONO, KD_HERCULES, KD_CGA, KD_EGA and KD_VGA.

The desired physical address is returned in *physaddr* (e.g. 0XB8000 for CGA).

The third element is an array of I/O addresses that the driver will allow operations on. These addresses are I/O port addresses on the video display controller. Some possible values that one might find here are 0x3B4, 0x3B5, 0x3D4 and 0x3D5.

Mapping the video memory

To access the video buffer you must request that the physical memory obtained by calling KDDISPTYPE be mapped into your address space. This is accomplished with the display driver *ioctl(2)* system call, KDMAPDISP. In using KDMAPDISP, the argument passed in the call is a pointer to the structure:

```
struct kd_memloc {
    char    *vaddr;       /* virtual address to map to */
    char    *physaddr;    /* physical address to map from */
    long    length;       /* size in bytes to map */
    long    ioflg;        /* enable i/o addresses if set */
};
```

The first element of the array, *vaddr*, must be a page aligned pointer to a physical area of memory. This will point to the area of memory within your address space where the video buffer will be made accessible to you. You must use *malloc*, or another allocation routine, to obtain this memory space.

The physical address, *physaddr*, is passed as input to KDMAPDISP. This is the address returned by the previous call to KDDISPTYPE.

Mapping example

To illustrate the mapping concept, consider the following code fragment:

```
/* The assumption here is that vaddr is a pointer to an array at least
 * 4096 bytes larger than needed to hold the mapped data.  The reason
 * for this is to ensure that when the pointer is page aligned that
 * the entire mapped area is addressable.  NOTE:  The desired area
 * could also be allocated using the alloc() functions. */
char scrmem[16384 + 4096];        /* define the memory array */
map_memory()
{
    struct kd_disparam parm_area, *kdp;
    struct kd_memloc map;
    char *vaddr;
    int disp;
    kdp = &parm_area;
    vaddr = scrmem;
    if((disp = open("/dev/video", O_RDWR | O_NDELAY)) < 0) {
        fprintf(stderr,"driver open failed, errno = %d\n, errno);
        exit(-1);
    }
    if(ioctl(disp, KD_DISPTYPE, kdp) < 0) {
        fprintf(stderr,"KD_DISPTYPE failed, errno = %d\n", errno);
        exit(1);
    }
    map.physaddr = kdp->addr;        /* set the video address */
    vaddr = (vaddr + 4095) & ~4095;  /* page align the address */
    map.vaddr = vaddr;               /* set the address */
    map.length = 16384;              /* set the screen size */
    map.ioflg = 1;                   /* enable io addresses */
    if(ioctl(disp, KDMAPDISP, &map) < 0)
    {
        fprintf(stderr,"KDMAPDISP failed, errno = %d\n",errno);
        exit(-1);
    }
}
```

Upon return from KDMAPDISP, *vaddr* points to an area of the mapped video buffer. Reads and writes to the memory at *vaddr* directly affect the information displayed on the screen.

To unmap the display buffer from user memory, the KDUNMAPDISP *ioctl* is provided. There are no arguments to KDUNMAPDISP and it may be called as follows:

```
if(ioctl(disp, KDUNMAPDISP, 0) < 0) {
    fprintf(stderr, "KDUNMAPDISP failed, errno = %d\n",
        errno);
    exit(-1);
}
```

The Section "Comprehensive video programming example", contains additional examples of accessing video

memory. The routine *get_display_info* illustrates the use of KDDISPTYPE. The routine *map_video_screen* similarly illustrates the use of KDMAPDISP.

Relationship to MAPCONS

An investigation of the list of *ioctl(2)* system calls show several alternative ways to do a number of the functions. Some of the calls exist to allow compatibility with programs previously written for XENIX. A specific question may arise here regarding the use of the MAPCONS call (i.e. *ioctl(disp, MAPCONS, 0);*). The MAPCONS call establishes a direct pointer to video memory and seems easier to use than the method described here which requires allocating the memory in your own program space.

However, the use of MAPCONS locks the Virtual Terminals and the end-user cannot switch Virtual Terminals until you specifically exit your application and free up the terminal.

Using KDMAPDISP in conjunction with process mode (see following sections), keeps your application well-behaved in a Virtual Terminal environment.

3.5.4 Graphics mode

3.5.4.1 Addressing graphics mode video memory

Memory allocation for each of the adapters, when used in graphics mode, is displayed in the following table:

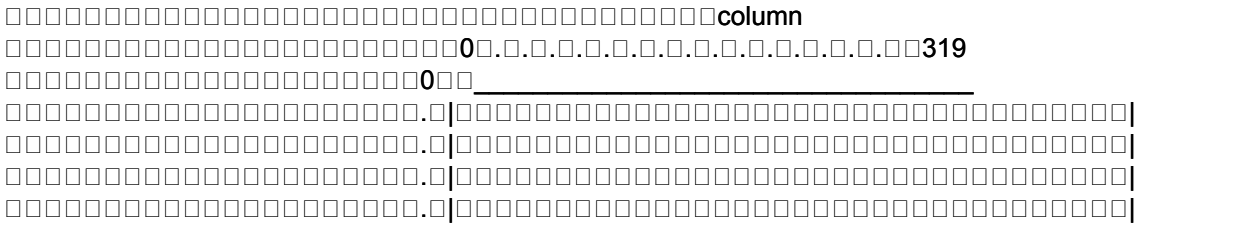
Mode	Type	Resolution	Colours	Address	Type
4	Graphics	320 x 200	4	0XB8000	CGA
5	Graphics	320 x 200	4 grey	0XB8000	CGA
6	Graphics	640 x 200	2	0XB8000	CGA
D	Graphics	320 x 200	16	0XA0000	EGA
E	Graphics	640 x 200	16	0XA0000	EGA
F	Graphics	640 x 350	2	0XA0000	EGA
10	Graphics	640 x 350	4	0XA0000	EGA
10	Graphics	640 x 350	16	0XA0000	VGA
11	Graphics	640 x 480	2	0XA0000	VGA
12	Graphics	640 x 480	16	0XA0000	VGA
13	Graphics	320 x 200	256	0XA0000	VGA

Table 24: Video modes for CGA/EGA/VGA

Accessing the video buffer when in graphics modes differs significantly from the mechanism used in text mode. In text mode characters displayed on the screen are stored in the video buffer as a data byte and an associated attribute byte; two bytes correspond to a single fixed screen position. Accessing this information is fairly simple.

Graphics modes do not operate on a character level, but on a pixel level. Graphics screen addressability is on a row/column basis; however, the row and column correspond to the pixel coordinates on the displayed screen. As an example, consider a resolution of 320x200. This resolution implies that there are 200 rows of 320 columns.

Pixel coordinates



```
□□□□□□□□□□row□□□□□□□□.□|□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□□□□□□□□□□□□□□□□.□|□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□□□□□□□□□□□□□□□□.□|□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□□□□□□□□□□□□□□□□.□|□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□□□□□□□□□□□□□□□□.□|□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□□□□□□□□□□□□□□□□.□|□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□□□□□□□□□□□□□□199□|__________________________________|
```

Accessing a pixel within the video RAM is not as straight forward as characters in text modes. As graphics modes vary, so do the ways that the information is stored within memory. A four color graphics mode requires two bits to store the four discrete color values. The remaining six bits within a byte are available to store pixel values for other screen locations. In this way, a single byte can store four pixel's worth of data in four color graphics modes. As the number of available colors increases, so does the memory required to store the additional information.

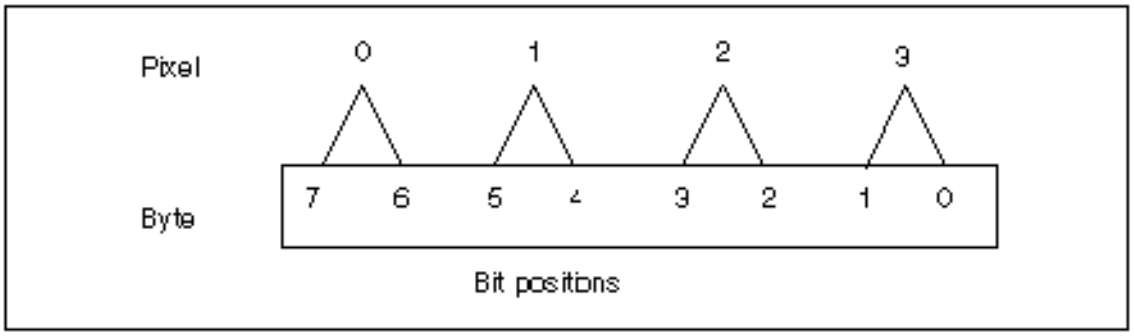


Figure 9: Accessing pixels in video RAM

A programmer needs to be concerned not only with pixel addressing within a byte, but also with which screen row is being addressed. In CGA graphics modes for instance, even rows (scan lines) are within one area of memory (0XB8000) and odd scan lines start at (0XB8000 + 0X2000) or 0XBA000. Putting both concepts together, the following address map results:

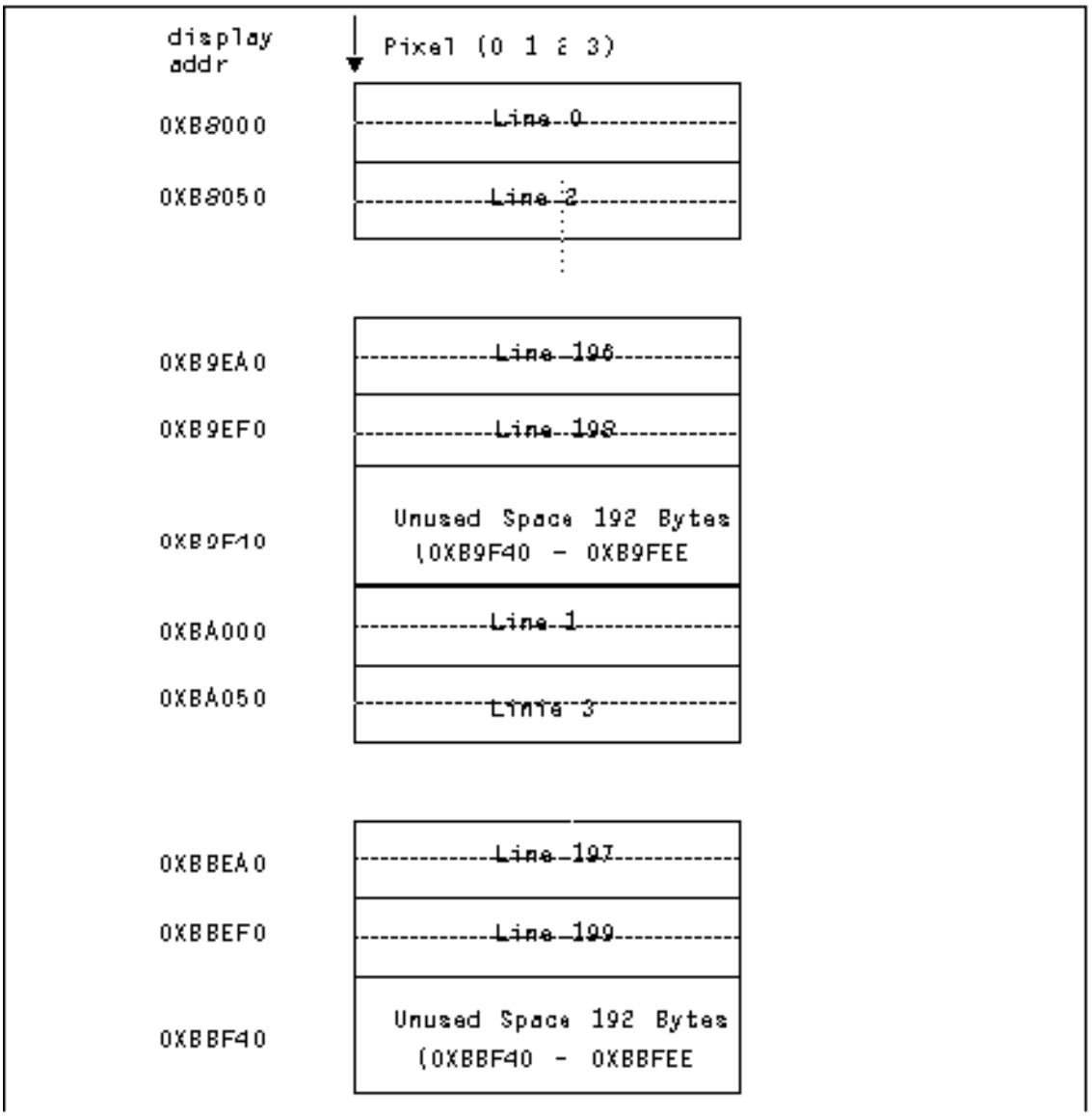


Figure 10: Pixel and screen row addressing

The above map shows that there is an area of memory at 0xB9f40 and 0xBAF40 that is unused. This is because only 8000 bytes are needed to represent 100 scans lines (80 bytes per row * 100 rows). Because 0x2000 bytes are allocated for this area (8192 bytes), this leaves 8192 - 8000 = 192 bytes of unused space. To properly access a particular byte within mapped memory (for modes 4 & 5) , the following formula could be used:

address□=□0xB8000□+□0x2000□*□row□%□2□+□row/2□*□0x50□+□column/4

It is important to note that this discussion is limited to CGA modes 4 and 5. After this section we will give examples for EGA addressing (see Addressing video memory in EGA graphics modes). These should give you a sufficient basis for developing your own memory addressing algorithms.

To access the last pixel within the last row of the above display (row = 199, column = 319) the formula above would yield

0xB8000□+□0x2000□*□0xC7□%□2□+□0xc7□/□2□*□0x50□+□0x13F□/□4□=
0xBA000□+□0x1EF0□+□0x4F□=□□0xBBF3F

Since the pixel is embedded within the byte at 0xBBF3F an additional step is necessary to extract the value.

This could be accomplished by masking out the required bits within the byte. One method for accomplishing this would be as follows:

value = byte & (0xA0 >> (column % 4) * 2)

In the above case, the formula would yield

value = byte & (0xA0 >> (319 % 4) * 2)

or

value = byte & (0xA0 >> 6) = byte & 0x3

This is exactly what we want since column 319 should be in the lower two bits of the byte at 0xBBF3F.

The following programming example illustrates one method of writing a pixel of information. It accepts arguments: *row*, *col* and *color*. It is assumed that the driver has already been opened and that *screen* is a page-aligned pointer to an already mapped video buffer.

```
/* mask and shift defines.  These are needed to place the color
 * information into the correct place within the written byte.
 */
unsigned int bmask[4] = {0xff3f, 0xffcf, 0xfff3, 0xfffc};
int color_shift[4] = {6, 4, 2, 0};
int write_pixel(row, col, color)
int row, col, color;
{
    int     index;            /* screen byte index */
    unsigned int mask;
    char *sptr;               /* screen memory pointer */
    sptr = screen;            /* set pointer to screen area */
    /* Find the correct position in the screen memory
     * The video memory is set up with odd/even
     * rows residing at different (non)contiguous memory addresses.
     * Because of this, we need to ensure that the starting address
     * of the byte we need to access is indeed correct.
     */
    index = row * 40 + col/4;
    /* Now decide whether the row was even or odd.  If the row
     * is odd, use (0xB8000 + index + 8152).  If the row
     * is even, use (0xB8000 + index).  This means
     * that the first odd row starts at (0xB8000 + 40 + 8152 = 0xBA000)
     * In short, the calculations ensure that the 8192 byte offset
     * between 0xb8000 and 0xBA000 is handled correctly.
     */
    if(row % 2)
        index += 8152;        /* bank 2 if odd */
    color <<= color_shift[col % 4];   /* shift color information */
    mask = bmask[col % 4];            /* select the right mask */
                                      /* buffer write */
    *(sptr + index) = color | *(sptr + index) & mask;
    return(SUCCESS);
}
```

Addressing video memory in EGA graphics modes

EGA supports additional modes with higher resolution than those of CGA. To support the additional modes more video memory is needed to store the pixel values. An EGA video controller supports up to 256KB of video buffer. (This is also the case for the VGA). This buffer is physically addressable within the address spectrum as a single 64KB block of physical memory. This is accomplished by overlaying the four 64KB blocks (planes) of memory at the same physical address. Selection of a plane is controlled by registers within the video controller.

As an example, consider a resolution of 320 x 200 with 16 colors. This would require 64Kb * 4bits or 32KB of RAM. The pixels are situated within a byte such that each physical pixel location on the screen maps to one bit within an addressed byte. This however, would only allow two colors to be displayed. To support 16 colors, four bits are needed. All four bits are mapped to the same location in a different plane. One way to access all

four bits of color data for a selected pixel location might be to read the same location in video memory four consecutive times while selecting the appropriate plane within the video controller (using the "Read Map Select Register" command). A graphic example is provided below:

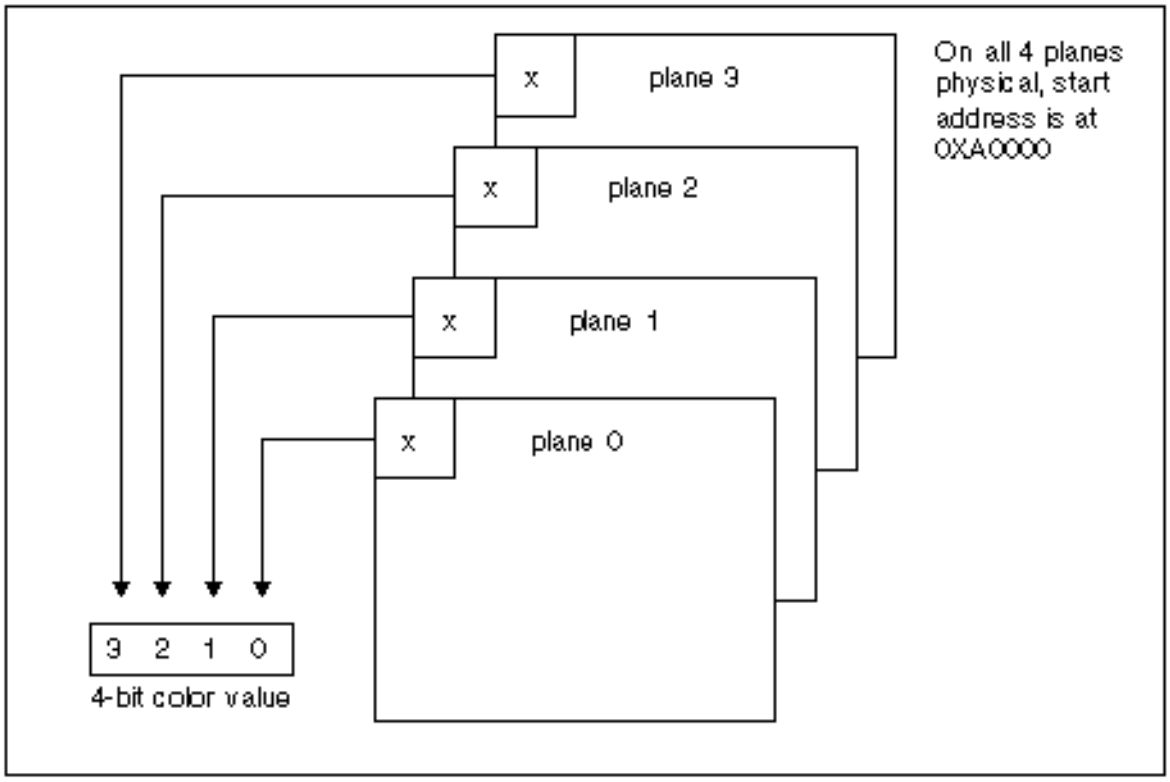


Figure 11: 16-color support

The above 4 bit color value would be used by the video controller hardware to represent the 16 colors for a selected pixel.

Mode F (640x350, two color) requires two bits to store a single pixel's worth of data. The first bit is the video bit (on/off) and the second bit is really an attribute (intensity). This information is stored in two memory planes; plane 0 and 2. As with all native EGA modes, pixels are stored consecutively within memory in a manner similar to alphanumeric modes for CGA. To access the pixel data for a particular x/y coordinate, the offset is a linear index from the start of the video buffer.

As an example, consider y = 0, x = 30. The index from the start of video RAM for this would be 0xA0003. This is computed by taking the start of the video buffer (0xA0000), adding the row displacement (80 bytes per row - here it is 0) and finally adding the x coordinate divided by the number of bits per byte.

This can be summarized by the following computation:

0xA0000 + y * BYTES_PER_LINE + BYTES_OFF_SET_WITHIN_LINE

or

0xA0000 + y * 80 + (30 Bytes/8 Bits Per byte)

or

0xA0000 + 0 + 3 = 0xA0003

Two things to remember are that the x coordinate in this example after the division is the integer value 3. The remainder (6) is the bit offset within the addressed byte. This implies a masking operation to extract the desired bit. The second thing to remember is that the intensity bit still needs to be retrieved from bit plane number 2 (at the same address).

VGA-only addressing modes

Modes D, E and F are VGA only modes. Addressing of video memory in these modes differs only because of the hardware differences in the video controller. The driver interface to VGA video hardware is identical to that for CGA and EGA. Rather than duplicate what has already been described for CGA and EGA, the reader is referred to readily available VGA reference manuals for information specific to VGA addressing and operation.

Mode switching

Changing from text mode to graphics mode is a simple *ioctl(2)* system call. For example, the following will change the display from text mode to 320 columns x 200 rows CGA graphics Mode 5:

```
{
...
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□<□0)□{
□□□□□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno:□%d\n",□errno);
□□□□□□□□□□□□□□□□exit(1)
□□□□□}
□□□□□if□(ioctl(disp,□SW_CG320,□0)□!=□1)□{
□□□□□□□□□□□□□□□□fprintf("stderr,□"Unable□to□select□SW_CG320□graphics□mode,
□□□□□□□□□□□□□□□□□□□□□□□□□□□□errno:□%d\n",errno);
□□□□□□□□□□□□□□□□exit(1)
□□□□□}
...
}
```

The video display buffer is accessed by opening */dev/video*. Use the KDSETMODE *ioctl* to reset the display back to text mode when done:

```
if□(ioctl□(disp,□KDSETMODE,□KD_TEXT)□<□0)□{
□□□□□fprintf(stderr,□"KDSETMODE□failed,□errno:□%d\n",errno);
}
```

3.5.5 Accessing video controller registers

Control Register programming is valuable in both text and graphics modes. In text mode, for example, it allows you to select alternate character sets.

To access the CRT controller registers, a two step procedure is needed. During the first step, an I/O location is written with a value corresponding to the controller register desired. For example, 0x3D4 is one of these key I/O locations. Some VDCs have more than one such location and the location to use may differ for monochrome and color adapters. Because of this, you should use the *outb* function call to access the register. The step of writing to this location tells the controller hardware which register's contents to make ready. To read the desired controller register, a second I/O operation (read) is directed to another location, typically 0x3D5. The byte of data returned is the value of the register requested. The following table lists some of the registers and the values that are used to select them. The example below (and the more extensive example in Appendix F) illustrates performing such an operation.

Note that in conditions in which a VGA card is doing monochrome emulation, use of the 3d4/3d5 registers should be replaced by 3b4/3b5. An application can determine whether to use 3d or 3b with the following code:

```
int□regbase□=□0x3b0;□□□/*□default□to□3b0□*/
if□(inb(0x3cc)&0x01)
□□□regbase□=□0x3d0;□□□□/*□3cc□is□CRT□status□byte.□The□*/
□□□□□□□□□□□□□□□□□□□□□□□□□/*□least□significant□bit□is□set□if□VGA□*/
□□□□□□□□□□□□□□□□□□□□□□□□□/*□is□in□□color□mode□and□not□mono-□*/
□□□□□□□□□□□□□□□□□□□□□□□□□/*□chrome□emulation□*/
```

Register	Register name
0x00	Horizontal Total
0x01	Horizontal displayed End
0x02	Start Horizontal Blank
0x03	End Horizontal Blank
0x04	Start Horizontal Retrace
0x05	End Horizontal Retrace

0x06	Vertical Total
0x07	Overflow
0x08	Preset Row Scan
0x09	Maximum Scan Line Address
0x0A	Cursor Start
0x0B	Cursor End
0x0C	Start High Address
0x0D	Start Low Address
0x0E	Cursor Location High
0x0F	Cursor Location Low
0x10	Vertical Retrace Start
0x11	Vertical Retrace Low
0x12	Vertical display End
0x13	Offset
0x14	Underline Location
0x15	Start Vertical Blanking
0x16	End Vertical Blanking
0x17	Mode Control
0x18	Line Compare

Table 25: VGA CRT controller registers

All of the registers above are read/write in a VGA based video system. In EGA and CGA systems, not all registers are both read and write; refer to a video technical reference manual for details.

Using registers for efficiency

Many applications will need more efficient ways of quickly filling the video buffer than the byte by byte method used in the examples. These applications will use the most efficient hardware read and write modes when updating the display. Doing so requires setting up certain hardware registers described in the Hardware Technical Reference Manual. Before these hardware registers may be accessed, an internal I/O flag must be set. One way is to set the *ioflg* variable of a *kd_memloc* structure to 1 before mapping the display using *KDMAPDISP*. An equivalent way is to explicitly set the I/O enable flag via the *KDENABIO ioctl(2)* system call. Once the flag is set, data may be moved in and out of the registers a byte at a time by using the *inb* and *outb* subroutines defined in the */usr/include/sys/inline.h* header file. The first argument of the call is the hardware register address. The second is the data byte to be moved into or out of the register.

To prevent errant or malicious programs from leaving a shared display in an inconsistent state upon exit, the KDDELIO *ioctl* is provided. It restricts access to the specified hardware registers by removing them from an internal list of valid register port addresses. It is ordinarily used by the system administrator and requires superuser privileges. These registers may later be restored to the list by using the KDADDIO *ioctl*.

Register programming example

The following code fragment demonstrates how to make use of the *ioctl(2)* system call interface to change the shape of the cursor. This example is only illustrative; you can change the cursor shape simply by issuing an escape sequence: *ESC [c0* changes the shape to an underscore; *ESC [c1* changes it to a block.

Example of video register access

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

```
#include□<sys/types.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/inline.h>
#include□<errno.h>
int□□disp;
□□/*□This□program□tests□the□KDENABIO□and□KDDISABIO□ioctl(2)□system□call.
□□□□□□In□order□to□have□an□easily□visible□result,□the□test□changes□the□cursor.
□□□□□□Depending□on□the□current□cursor□type,□the□cursor□will□shift□between□an
□□□□□□underscore□and□a□block.□□This□test□should□only□be□run□on□an□EGA□or□VGA
□□□□□□controller.□*/
main()
{
□□□□□struct□kd_vdctype□vdcinfo;
□□□□□unchar□□□□□□cur_start,□cur_end,□new_val;
□□□□□int□regbase□=□0x3b0;
□□□□□printf("\033c")
□□□□□fflush(stdout);
□□□□□if□(ioctl(0,□KIOCINFO,□0)□!=□(('k'□<<□8)□|□'d'))□{
□□□□□□□□□□□□fprintf(stderr,□"These□tests□are□meaningless□if□run□on");
□□□□□□□□□□□□fprintf(stderr,□"a□TTY□other□than□a□graphics□display!!\n")
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno□%d\n\n"
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□if□(ioctl(disp,□KDENABIO,□0)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"KDENABIO□failed:□errno□%d\n\n",□errno
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□}
□□□□□if□(vdcinfo.cntlr□==□KD_VGA□||□(inb(0x3cc)&0x01))
□□□□□□□□□□□□regbase□=□0x3d0;
□□□□□□□/*Inform□the□adapter□that□you□will□read□register□0x0a□(Cursor\ Start)*/
□□□□□outb(regbase+4,□0x0a);
□□□□□□□□□□/*□And□read□it□*/
□□□□□cur_start□=□inb(regbase+5);
□□□□□□□□□□/*□Similarly,□read□Cursor□End□*/
□□□□□outb(regbase+4,□0x0b);
□□□□□cur_end□=□inb(regbase+5);
□□□□□if□(cur_start□==□(cur_end□-□1))□{□□□□□□□□□□□□□□□/*□line□cursor□*/
□□□□□□□□□□□□printf("The□cursor□is□an□under-score□");
□□□□□□□□□□□□fflush(stdout);
□□□□□□□□□□□□new_val□=□0x00;□□□□□□□□□□□□□□□□□□□/*□set□value□for□block□*/
□□□□□}
□□□□□else□{□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□/*□assume□block□*/
□□□□□□□□□□□□printf("The□cursor□is□a□block□");
□□□□□□□□□□□□fflush(stdout);
□□□□□□□□□□□□new_val□=□cur_end□-□1;□/*□and□set□value□for□line□*/
□□□□□}
□□□□□sleep(5);
□□□□□printf("\nChanging□the□cursor□now□")
□□□□□fflush(stdout);
□□□□□□□□□□/*□Change□the□Cursor□Shape□*/
□□□□□outb(regbase+4,□0x0a);
□□□□□outb(regbase+5,□new_val);
□□□□□sleep(5);
□□□□□outb(regbase+4,□0x0a);
□□□□□outb(regbase+5,□cur_start);
□□□□□errno□=□0;
□□□□□if□(ioctl(disp,□KDDISABIO,□0)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"KDDISABIO□failed:□errno□%d\n\n",□errno
```

```
□□□□□□□□□□□□exit(1);
□□□□□□}
□□□□□printf("\n\nThis□test□fails□if□the□cursor□did□not□change.\n
□□□□□exit(0);
}
```

The example illustrates the sequence of steps necessary to access video controller registers. Such an example may be extended to include access to other registers within the video controller. The example in Appendix F is more complete in terms of how an actual program might be structured.

The KDENABIO *ioctl* informs the driver that the user program will perform I/O to the video controller. The KDDISABIO *ioctl* informs the driver that the I/O capability should be disabled. The KDADDIO *ioctl* adds an address to the allowable I/O port list, and the KDDELIO *ioctl* removes an address. For instance, these routines could be used to add/delete I/O addresses 0x3C4 and 0x3C5. These addresses correspond to the EGA/VGA sequencer registers. Once the addresses are added, a user program could use them to access the 6 sequencer registers. The following example illustrates how one might use the KDADDIO and KDDELIO *ioctl*s.

```
extern□int□disp;□/*□driver□file□descriptor□*/
{
□□□□□if(ioctl(disp,□KDADDIO,□(unsigned□short)□0x3C4)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,"KDADDIO□failed,□errno□=□%d\n,□errno);
□□□□□□□□□□□□exit(-1);
□□□□□}
□□□□□if(ioctl(disp,□KDADDIO,□(unsigned□short)□0x3C5)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,"KDADDIO□failed,□errno□=□%d\n,□errno);
□□□□□□□□□□□□exit(-1);
□□□□□}
}
Deletion of addressable ports is similar and is demonstrated below:
extern□int□disp;□/*□driver□file□descriptor□*/
{
□□□□□if(ioctl(disp,□KDADDIO,□(unsigned□short)□0x3C4)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,"KDADDIO□failed,□errno□=□%d\n,□errno);
□□□□□□□□□□□□exit(-1);
□□□□□}
□□□□□if(ioctl(disp,□KDADDIO,□(unsigned□short)□0x3C5)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,"KDADDIO□failed,□errno□=□%d\n,□errno);
□□□□□□□□□□□□exit(-1);
□□□□□}
}
```

3.5.6 Using Virtual Terminals

Virtual Terminals (VTs) are designed to enhance the interfacing capabilities of Reliant UNIX System V/386. They represent the next evolutionary step in the use of terminals for graphics applications. The first advance was allowing a single user to develop a graphics application. Over the last few years, windowing systems have become widely available. In these systems, several users can develop graphics applications under the control of a windowing system. Reliant UNIX 5.43 allows several windowing systems to operate simultaneously. It is not necessary to stop one application before another starts.

In this respect the Virtual Terminal capability of Reliant UNIX 5.43 analogous to your television set; changing Virtual Terminals is like switching channels. The programs continue on the other channels and you tune in to them. The analogy breaks down in several very important aspects:

- In a VT environment, a program that is switched away can put itself to sleep so that nothing happens until it is again activated (i.e. there is no chance that the user can miss any output).
- A program can hog the system and insist that it not be switched away.
- An application program can create and manage Virtual Terminals independent of the end-user. That is, the application can switch terminals based on purely functional requirements.

This section describes how to use Virtual Terminals in these respects:

- Use by end users.
- Writing graphics applications that are 'aware' of VTs and are well-behaved.

- Writing VT management applications.

3.5.6.1 Use of Virtual Terminals

New virtual terminals are created by a particular 'hot key' sequence once *vtlmgr* has been invoked, specifically:

ALT - SYS-REQ key

where *key* is either a function key whose number corresponds to the number of the VT to switch into (F1, F2, ...) or, if virtual terminals have already been created with *vtlmgr* or *newvt*, one of the following letters:

Key	Interpretation
h	home VT
n	next VT
p	previous VT
f	force a switch to a VT

Table 26: Function keys for calling Virtual Terminals

The *f* key is used only when a user discovers that the current VT is essentially locked up or stuck in graphics mode. This will cause the *kd* driver to reset the VT to a sane text state and kill all processes associated with the VT.

The user can control how many VTs are available by setting a parameter in the file */etc/default/workstations*. VT 0 - 8 are configured by default and the default keyboard map makes up to 13 VTs available (i.e., the user can readily define an additional 4 VTs within the default settings). The default VTs are the home terminal and one corresponding to each function key. An application can make two more available to the end-user (by reprogramming the keyboard map), or can reserve the last two for programmatic use only, making 15 VTs in all.

The end-user also needs to be aware that processes that are no longer visible may still be continuing. Standard output is directed to the current VT screen. For example, a user can issue a cat command on one VT and can then switch to another VT to start an application and another to do an edit. The cat output can be lost unless the user initially redirects that output to a file (if the virtual terminal scrolls the data off the screen).

3.5.6.2 Programming features

VTs are a kernel maintained resource. The kernel keeps all VTs for a particular terminal on a circular queue. The 0th VT is special and is activated if all other VTs on that device are closed. Any process desiring Virtual Terminals must compete with all other processes using a Virtual Terminal associated with the same minor device.

Programs that use VTs must include the header file */usr/include/sys/vt.h*. The *kd* driver supports the Virtual Terminal feature in the following ways:

- It allows switching between open Virtual Terminals.
- It maintains the file */dev/vtmon* for signalling user requests for VT activation,
- It responds to a series of controlling *ioctl* commands.

The *ioctl* commands used to control VT operation are:

Command	Function
VT_SETMODE	Sets the VT mode to automatic or process controlled.
VT_GETMODE	Gets the current VT mode.
VT_RELDISP	Releases, refuses to release, or acquires the display.
VT_WAITACTIVE	Waits until the specified VT becomes active.
VT_OPENQRY	Returns the number of the next available VT.

VT_ACTIVATE	Activates the specified VT.
VT_GETSTATE	Returns the active VT number and list of open VTs,
VT_SENDSIG	Sends a specified signal to open VTs owned by the process.

Table 27: ioctl commands used to control Virtual Terminal operation

The first four commands are required for the development of well-behaved graphics applications. The remaining four commands are only required when you want to develop an application that manages multiple VTs.

Note: *ioctl*s done in the background on a VT may cause the process to block.

VT modes of operation

There are two modes of operation for Virtual Terminals. The first mode is the automatic mode (VT_AUTO) of operation. This is the simplest case and the default case. In automatic mode the application is not made aware of the end-user's requests to switch away from or back to the current VT. This means that any output in process while the user is switched away may be lost. The only option at your disposal is to issue the VT_WAITACTIVE command before every output statement. This command causes the program to suspend operation until it is using the currently active VT. This is, at best, tedious.

The second mode is the process control mode, or *process mode*, VT_PROCESS. This mode allows you to synchronize your application with other processes that are using VTs. When you set the mode to VT_PROCESS you assume the responsibility of accepting and relinquishing use of the VT.

3.5.6.3 Writing well-behaved programs

The most common circumstance is that you want to develop a graphics application that will run in a VT environment. You want to be responsible for its running in that environment and you want to take minimal responsibility for that aspect of your program.

This is how to set up your program to run in this manner:

1. Make sure you use the KDMAPDISP command to manage your use of display memory.
2. When you are ready to initiate the use of the terminal, issue a VT_GETMODE command. This command passes a structure to the driver. The driver fills the structure with information, but you are only interested in the current mode. If the mode is already VT_PROCESS, there is an error condition. You cannot go into process mode from process mode.
3. Assuming you can continue, issue a VT_SETMODE command. This command passes the mode structure into the driver. The VT_SETMODE command accomplishes four things:
   - It establishes VT_PROCESS mode.
   - It defines the signal to be received when the end-user requests to switch to another VT.
   - Normally, when your VT is not active your program can continue to execute. You can specify in this command that even though your program can continue to execute, it will not perform any output until the VT is again active. That is, it will hang when it encounters an output instruction.
4. Write signal processing routines to process each of the signals defined by VT_SETMODE.
   - Relinquishing control

     Once a request to switch out of your VT has been issued, you must respond within ten seconds. The VT_RELDISP command is the response vehicle. If you issue the command with a zero argument you are refusing to give up control (and the end-user will get a "beep" at the terminal). A non-zero argument says "okay to switch now". Before you authorize the switch, you need to save the display mode (unless you explicitly know it), restore the adapter to the default text mode, and save the status of any registers you have explicitly changed. You also issue a *KDUNMAPDISP* to cut the linkage between the video memory and your memory. The last thing you do is issue the *VT_RELDISP* command.
   - Regaining control

     When the signal indicates that you have regained control you do the opposite tasks; you first issue the

VT_RELDISP command with an argument of *VT_ACKACQ* (ACKnowledge ACQuisition); then reinitialize the adapter mode; re-modify any specially set registers and re-establish the connection between your memory and the video memory.

Example of a well-behaved graphics application

The example that follows illustrates the techniques to develop a well-behaved graphics application.

```
#include□<stdio.h>
#include□<fcntl.h>
#include□<sys/types.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/vt.h>
#include□<sys/signal.h>
#include□<sys/inline.h>
#include□<errno.h>
unchar□□□□□*Tdisplay,
□□□□□*Display;
int□disp;
struct□kd_memloc□memloc;
struct□kd_disparam□□□□kd_param;
struct□modes□{
□□□□□char□□m_name[20];□□□□□/*□mode□name□*/
□□□□□int□□□m_value,□□□□□□□□□/*□ioctl□value□*/
□□□□□□□□□□□□m_layout,□□□□□□□□/*□memory□layout□*/
□□□□□□□□□□□□m_cnt,□□□□□□□□□□□/*□count□for□drawing□*/
□□□□□□□□□□□□m_len;□□□□□□□□□□□/*□length□of□memory□to□map□*/
}□Modes□=□{
□□□□□□□□□□□□□"SW_CG320",
□□□□□□□□□□□□□SW_CG320,
□□□□□□□□□□□□□1,
□□□□□□□□□□□□□8000,
□□□□□□□□□□□□□16384
};
/*
This□routine□fills□the□screen□with□a□simple□pattern.□□CGA□mode□was□chosen
because□it□is□common□to□all□of□the□color□adapters.
*/
redraw_scrn()
{
□□□□□int□cnt1;
□□□□□if□(!Modes.m_layout)
□□□□□□□□□□□□□return(0);
□□□□□errno□=□0;
□□□□□for□(cnt1□=□0;□cnt1□<□Modes.m_cnt;□cnt1++)□{
□□□□□□□□□□□□□*(Display□+□cnt1)□=□0xa5□;□□□□/*□0xa5□is□random□pattern□*/
□□□□□□□□□□□□□*(Display□+□8192□+□cnt1)□=□0xa5;
□□□□□}
}
/*
This□routine□is□entered□when□the□user□terminates□the□session.□□It□unmaps□the
display□and□resets□the□mode□before□exiting.
*/
sigintr()
{
□□□□□if□(ioctl□(disp,□KDUNMAPDISP,□0)□==□-1)□{
□□□□□□□□□□□□□fprintf(stderr,□"KDUNMAPDISP□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□if□(ioctl□(disp,□KDSETMODE,□KD_TEXT)□==□-1)□{
□□□□□□□□□□□□□fprintf(stderr,□"KDSETMODE□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□exit(0);
```

```
}
/*□This□routine□is□entered□when□there□is□a□request□to□switch□to□another
□□□Virtual□Terminal.
□□□The□process□is□to□unmap□the□display,□restore□the□adapter□to□text□mode
□□□and□then□allow□the□switch.
*/
sigusr1()
{
□□□□□if□(ioctl□(disp,□KDUNMAPDISP,□0)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"KDUNMAPDISP□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□if□(ioctl□(disp,□KDSETMODE,□KD_TEXT)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"KDSETMODE□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□if□(ioctl□(disp,□VT_RELDISP,□1)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_RELDISP□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□sigrelse(SIGUSR1);
}
/*This□routine□is□entered□when□the□user□re-activates□this□VT.□□The□RELDISP□is
□□issued□first,□to□acknowledge□that□the□program□is□back.□□You□then□set
□□graphics□mode□and□re-map□display□memory.
*/
sigusr2()
{
□□□□□if□(ioctl□(disp,□VT_RELDISP,□VT_ACKACQ)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_RELDISP□ack□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□if□(ioctl□(disp,□SW_CG320,□0)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"SW_CG320□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□if□(ioctl(disp,□KDMAPDISP,□&memloc)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"KDMAPDISP□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□redraw_scrn();
□□□□□sigrelse(SIGUSR2);
}
/*The□main□program□sets□up□the□interrupt□signal□routines□and□draws□the
□□initial□screen.
*/
main()
{
□□□□□int□□□□□□□□□□cnt,□ch,□mask□=□0;
□□□□□struct□vt_mode□□□vtmode;
□□□□□if□(ioctl(0,□KIOCINFO,□0)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"These□tests□are□meaningless□if□run□on");
□□□□□□□□□□□□fprintf(stderr,□"a□TTY□other□than□a□graphics□display!!\n")
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno□%d\n\n"
□□□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□fprintf(stdout,"This□VT□will□be□put□into□graphics□mode,\n")
□□□□□fprintf(stdout,"a□picture□will□be□drawn.□We□also□put□the□VT\n")
□□□□□fprintf(stdout,"in□process□mode,□so□you□can□VT□switch□to□other\n")
□□□□□fprintf(stdout,"VTs.□Upon□switching□back,□the□screen□should\n")
```

```
□□□□□fprintf(stdout,"be□repainted.□□This□tests□VT_SETMODE, KDDISPTYPE,\n")
□□□□□fprintf(stdout,"KDMAP/UNMAP/DISP,VT_RELDISP,SW_320CG□graphics□mode.\n")
□□□□□fprintf(stdout,"□and□text□mode\n\nThis□test□should□fail□if□you\n
□□□□□fprintf(stdout,"do□not□have□a□CGA/EGA/VGA□adapter\n\n"
□□□□□fprintf(stdout,"Hit□[DEL]□to□quit.\n\n"
□□□□□sleep(10);
□□□□□□□□/*□Make□sure□allocated□memory□is□page-aligned.□*/
□□□□□Tdisplay□=□(unchar□*)malloc(64□*□1024□+□4096);
□□□□□Display□=□(unsigned□char□*)((unsigned)(Tdisplay□+□4095)□&□~4095);
□□□□□□□□/*□Make□sure□we□are□not□already□in□process□mode.□*/
□□□□□if□(ioctl(disp,□VT_GETMODE,□&vtmode)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_GETMODE□failed:□errno□%d\n",□errno)
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□if□(vtmode.mode□==□VT_PROCESS)□{
□□□□□□□□□□□□fprintf(stderr,□"VT□is□already□in□VT_PROCESS□mode\n")
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□□□□/*□Set□up□process□mode□and□set□up□the□interrupt□handling structure.
□□□□□□□□□□□□In□this□case,□forced□switch□behavior□will□be□the□same□as normal
□□□□□□□□□□□□release□behavior.□*/
□□□□□vtmode.mode□=□VT_PROCESS;
□□□□□vtmode.relsig□=□SIGUSR1;
□□□□□vtmode.frsig□□=□SIGUSR1;
□□□□□vtmode.acqsig□=□SIGUSR2;
□□□□□if□(ioctl(disp,□VT_SETMODE,□&vtmode)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_SETMODE□failed:□errno□%d\n",□errno)
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Set□the□display□type□to□CGA□for□widest□applicability.□*/
□□□□□if□(ioctl□(disp,□SW_CG320,□0)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"SW_CG320□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□/*□Get□the□memory□address□of□the□display.□*/
□□□□□errno□=□0;
□□□□□if□(ioctl(disp,□KDDISPTYPE,□&kd_param)□==□-1)□{
□□□□□□□□□□□□fprintf(stderr,□"KDDISPTYPE□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□/*□Set□up□the□structure□for□the□KDMAPDISP□command.□*/
□□□□□memloc.vaddr□=□(char□*)Display;
□□□□□memloc.physaddr□=□kd_param.addr;
□□□□□memloc.length□=□Modes.m_len;
□□□□□memloc.ioflg□=□1;
□□□□□/*□Map□the□Display.□*/
□□□□□if□(ioctl(disp,□KDMAPDISP,□&memloc)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"KDMAPDISP□failed,□errno□%d\n",□errno)
□□□□□□□□□□□□return(0);
□□□□□}
□□□□□/*□Draw□the□screen.□□Set□up□the□signals□and□wait.□*/
□□□□□redraw_scrn();
□□□□□sigset(SIGUSR1,sigusr1);
□□□□□sigset(SIGUSR2,sigusr2);
□□□□□sigset(SIGINT,sigintr);
□□□□□/*□User□must□hit□break□(e.g.,□'DEL')□to□interrupt□the□infinite□loop.□*/
□□□□□while□(1)
□□□□□□□□□□□□;
}
```

3.5.6.4 Programming to manage Virtual Terminal use

The next sub-sections illustrate different aspects of a single process managing other processes that use VTs.

Suppose a process needs to respond to given events in an indeterminate fashion. By choosing a Virtual Terminal the process can handle just about any I/O requirements. By setting VT_PROCESS and spawning

event handling processes attached to the VTs that are waiting upon VT activation (either because they issued a VT_WAITACTIVATE command or because they issued a VT_SETMODE requesting a hangup on output), the server process can act as an event gateway relinquishing access to the VT to the associated event handling client and regaining access when it is through.

The following is a simple shell level command that kills all of the active processes associated with the Virtual Terminals. The program examines the Virtual Terminals and based on their state, individually sends a SIGKILL to each open VT. It illustrates the use of the *ioctl* request VT_SENDSIG.

VT_SENDSIG allows a process to send a particular signal to any combination of Virtual Terminals. This is a powerful feature extending the standard Reliant UNIX operating System's signal handling capabilities, creating a user definable Virtual Terminal group that can collectively or selectively act upon any given signal. A controlling process can exploit this during initialization, error recovery and synchronization.

Programming to manage Virtual Terminal use

```
#include□<stdio.h>
#include□<string.h>
#include□<sys/types.h>
#include□<signal.h>
#include□<fcntl.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/vt.h>
#include□<sys/termio.h>
#include□<sys/stat.h>
#include□<errno.h>
#include□<varargs.h>
#define□KD_DRIVER□0x6b64□□□/*□"kd"□*/
#define□FOS_DRIVER□0x73□□□/*□Fiber□Optic□Station□driver□*/
#define□EQUAL□0
char□vtname[20];
main(argc,□argv)
int□□argc;
char□**argv;
{
□□□□□int□i,□disp,□ttype,□activeVTs,□vtActive;
□□□□□struct□vt_stat□□vtinfo;
□□□□□/*Determine□what□type□of□terminal□we□are□on.□□This□must□be
□□□□□a□graphics□terminal□such□as□the□console.□*/
□□□□□ttype□=□ioctl(0,□KIOCINFO,□0);
□□□□□if□((ttype□!=□KD_DRIVER)□&&□((ttype□&□0xff00)□>>□8)□!=□FOS_DRIVER){
□□□□□□□□□□□□fprintf(stderr,□"Must□be□run□on□a□graphics□display.\n");
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Open□the□driver.□*/
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□<□0)□{
□□□□□□□□□□□□printf("Cannot□open□/dev/video\n");
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Get□the□state□information□from□the□driver.□□This□will□tell□us
□□□□□□*□which□VT□is□the□active□one□as□well□as□the□signal□states□and
□□□□□□*□allocated□VTs.□*/
□□□□□ioctl(disp,□VT_GETSTATE,□&vtinfo);
□□□□□activeVTs□=□vtinfo.v_state;
□□□□□vtinfo.v_signal□=□SIGKILL;□□/*□select□kill□signal□*/
□□□□□/*□Now□check□the□state□of□all□VTs□and□only□kill□those
□□□□□□*□that□are□truly□active.□*/
□□□□□for□(i□=□0;□i□<□15;□i++)□{
□□□□□□□□□□□□printf("\t/dev/vt%.2d□",□i);
□□□□□□□□□□□□/*□Note□that□a□single□bit□within□"v_state"□is
□□□□□□□□□□□□*□being□set□for□the□ioctl(VT_SENDSIG).□□This
□□□□□□□□□□□□*□is□acceptable□since□we□have□saved□the□active□VT
□□□□□□□□□□□□*□value□in□"activeVTs".□*/
```

```
□□□□□□□□□□□□□□if□(vtinfo.v_state□=□activeVTs□&□(1□<<□i))□□{
□□□□□□□□□□□□□□□□□□if□(i□==□vtinfo.v_active)□□{
□□□□□□□□□□□□□□□□□□□□□□□□/*□The□physically□active□VT□will□be
□□□□□□□□□□□□□□□□□□□□□□□□*□killed□last;□therefore□save□the
□□□□□□□□□□□□□□□□□□□□□□□□*□VT□number□in□vtActive.□*/
□□□□□□□□□□□□□□□□□□□□□□□□printf("will□be□killed□last.\n");
□□□□□□□□□□□□□□□□□□□□□□□□vtActive□=□i;
□□□□□□□□□□□□□□□□□□□□□□□□continue;
□□□□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□□□□/*□□This□VT□is□allocated;□therefore,□send
□□□□□□□□□□□□□□□□□□*□it□a□kill□signal.□□This□has□been□selected
□□□□□□□□□□□□□□□□□□*□by□setting□the□signal□type□in□the□structure
□□□□□□□□□□□□□□□□□□*□pointed□to□by□vtinfo.□*/
□□□□□□□□□□□□□□□□□□printf("...killing.\n");
□□□□□□□□□□□□□□□□□□ioctl(disp,□VT_SENDSIG,□&vtinfo);
□□□□□□□□□□□□}
□□□□□□□□□□□□else
□□□□□□□□□□□□□□□□□□printf("not□in□use.\n");
□□□□□}
□□□□□□/*□Now□we□need□to□kill□the□physically□active□VT,
□□□□□□□*□the□number□of□which□has□been□saved□in□"vtActive".□*/
□□□□□if□(vtActive)□□{□/*□not□the□console□*/
□□□□□□□□□□□□printf("\t/dev/vt%.2d□will□be□killed□now",□vtActive);
□□□□□□□□□□□□sleep(3);
□□□□□□□□□□□□vtinfo.v_state□=□activeVTs□&□(1□<<□vtActive);
□□□□□□□□□□□□ioctl(disp,□VT_SENDSIG,□&vtinfo);
□□□□□}
□□□□□exit(0);
}
```

3.5.6.5 Virtual Terminal creation and application

The following example is a program to create new Virtual Terminal shells. When executed, the program spawns a new process which then becomes another instance of the user's login session. Once such a session is established, the user may switch between the different shells using the keyboard driver "CTL-ALT-SYSREQ" key sequences. This would allow a user to have multiple login sessions active on a single physical terminal (system console). Because the functionality of Virtual Terminals is part of the keyboard driver, this functionality is not supported on remote terminals.

The *kd ioctl*s used within the application are: KIOCINFO, VT_OPENQRY and VT_ACTIVATE. KIOCINFO is used to ensure that the controlling driver for the terminal is the keyboard driver. Such a test is necessary because remote terminal drivers do not possess Virtual Terminal capability.

VT_OPENQRY is used to ask the driver for an available VT slot. This slot allows the kd driver to associate a virtual screen with a process. All screen input and output from a process must be directed to an area of memory within the driver until that screen's contents are mapped to the physical display terminal. VT_OPENQRY returns a slot number that also corresponds to the VT id used to build the pathname for the terminal special device (i.e. */dev/vt01*, */dev/vt02*,...).

VT_ACTIVATE, allows you to specify the VT to be mapped to the physical display device. The VT number to be mapped is passed as an argument in the VT_ACTIVATE ioctl. This is one way to have the application control which screen to switch to. This functionality could be embedded as a menu item within a screen menu for instance.

Creating and using Virtual Terminals

```
#include□<stdio.h>
#include□<sys/types.h>
#include□<string.h>
#include□<signal.h>
#include□<fcntl.h>
#include□<ctype.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/vt.h>
```

```
#include <sys/termio.h>
#include <errno.h>
extern int errno;
main(argc, argv)
int argc;
char *argv[];
{
    int fd, disp;
    long vtno;                          /* Virtual Terminal id */
    char prompt[11];                    /* prompt string */
    char vtname[VTNAMESZ];              /* vt name character string */
    ushort ttype;                       /* driver identification */
    struct termio term;                 /* termio terminal parameters */
    /* Open the controlling terminal for the device. */
    if ((fd = open("/dev/tty", O_RDONLY)) == -1)
        exit(1);
    /* Determine the special device name for the terminal.
     * The returned value should be "kd" for keyboard driver.  This is
     * to ensure that the user is executing on the console and not on
     * a remote terminal. */
    if ((ttype = ioctl(fd, KIOCINFO, 0)) == (ushort)-1) {
        fprintf(stderr, "KIOCINFO failed, errno = %d\n", errno);
        exit(1);
    }
    if (ttype != 0x6b64) {  /* "kd" */
        fprintf(stderr, "cannot execute %s on remote terminals",
                argv[0]);
        exit(1);
    }
    /* Get the terminal parameters for this terminal.  see termio(7) */
    if (ioctl(fd, TCGETA, &term) < 0) {
        fprintf(stderr, "TCGETA failed, errno = %d\n", errno);
        exit(1);
    }
    /* The VT_OPENQRY ioctl will ask the driver for the next
     * available Virtual Terminal port, i.e., (00, 01, 02, ...); */
    if (ioctl(fd, VT_OPENQRY, &vtno) < 0) {
        fprintf(stderr, "VT_OPENQRY failed, errno = %d\n", errno);
        exit(1);
    }
    if (vtno < 0) {
        fprintf(stderr, "No vts available, errno = %d\n", errno);
        exit(1);
    }
    sprintf(vtname, "/dev/vt%02d", vtno);   /* setup vt path */
    close(fd);
    close(2);   /* close stderr */
    close(1);   /* close stdout */
    close(0);   /* close stdin  */
    /* Now the program will fork a child.  The child process will
     * inherit the new Virtual Terminal.  Once the fork succeeds,
     * the parent program can terminate since its job is done.  The
     * child will continue with its environment setup.  */
    if (fork())
        exit(0);    /* parent exits */
    setpgrp();                          /* set group id to uid */
    /* Now open the Virtual Terminal as stderr.  The Virtual Terminal
     * name will correspond to one of the special device files:
     *
     * /dev/vt01, /dev/vt02, ...           */
    if (open(vtname, O_RDWR) == -1)
        exit(1);
    ioctl(0, TCSETA, &term);    /* restore terminal parameters */
    /* Re-establish stdout, stderr.  */
```

```
□□□□□dup(0);
□□□□□dup(0);
□□□□□if□((disp□=□open("/dev/video",□O_RDWR))□==□-1)
□□□□□□□□□□□exit(1);
□□□□□/*□Set□Virtual□Terminal's□prompt□variable□PS1□to□make□easier
□□□□□□*□identification□for□the□user.□*/
□□□□□sprintf(prompt,"PS1=VT□%d>□",□vtno);
□□□□□putenv(prompt);
□□□□□□/*□Clear□the□screen.□*/
□□□□□fputs("\033c",□stdout);
□□□□□signal(SIGINT,□SIG_DFL);□□□□/*□ignore□interrupts□*/
□□□□□signal(SIGQUIT,□SIG_DFL);□□□/*□ignore□quit□*/
□□□□□□/*□The□following□ioctl□will□activate□the□new□vt□so□that□the□display
□□□□□□□*□will□be□refreshed□with□the□new□VT's□virtual□screen.□*/
□□□□□if(ioctl(disp,□VT_ACTIVATE,□vtno)□<□0)□{
□□□□□□□□□□□fprintf(stderr,"VT_ACTIVATE□failed\n");
□□□□□□□□□□□/*□Continue□(this□should□not□happen□but□is□not□fatal)□*/
□□□□□}
□□□□□/*□This□will□spawn□a□new□shell□for□the□Virtual□Terminal.□□In□some
□□□□□□*□applications□this□could□be□an□exec□of□application□program.□*/
□□□□□□if□(execl("/bin/sh",□"sh",□0)□==□-1)
□□□□□□□□□□□fprintf(stderr,□"exec□of□/bin/sh□failed,□errno□=□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□/*□Should□only□get□here□if□the□execl□fails.□In□this□case□sleep□5 seconds
□□□□□□*□to□let□the□user□see□the□error□on□the□new□Virtual□Terminal.□*/
□□□□□sleep(5);
□□□□□exit(1);
}
```

3.5.6.6 Determining VT state

An interesting aspect of the VT feature is that you can gain access to the state information of active VT sessions. For instance, you may wish to know which VTs are allocated and which VT is currently controlling the display screen. The following example demonstrates the use of the VT_GETSTATE *ioctl* and the information it returns.

The example provided first opens the controlling tty to return a file descriptor for the console driver. This implies that the command can only run from the console in its current incarnation. Once a connection to the driver has been established, the VT_GETSTATE *ioctl* is invoked and a structure containing three items is populated by the console driver. The remainder of the program then individually examines each bit in the *v_state* element to determine the state of each of the 15 possible VTs.

Determining Virtual Terminal state

```
#include□<stdio.h>
#include□<fcntl.h>
#include□<sys/types.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/vt.h>
#include□<errno.h>
int□□disp;
main(argc,□argv)
int□□argc;
char□*argv[];
{
□□□□□int□□□cnt;
□□□□□struct□□vt_stat□□vtinfo;□□□/*□structure□for□VT_GETSTATE□*/
□□□□□/*□Open□the□controlling□terminal.□□If□this□is□successful,□the\ device
□□□□□□*□is□automatically□a□proper□controlling□tty.□*/
□□□□□if((disp□=□open("/dev/video",□O_RDONLY))□<□0)□{
□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□exit(1);
□□□□□}
```

```
□□□□□/*□Get□the□video□state□information.□□□*/
□□□□□if□(ioctl(disp,□VT_GETSTATE,□&vtinfo)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_GETSTATE□failed:□errno□%d\n",□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*
□□□□□□*□Print□the□contents□of□the□vtinfo□structure.□□v_active
□□□□□□*□is□the□number□of□the□active□Virtual□Terminal.□□v_signal
□□□□□□*□is□the□signal□state□of□the□process□controlling□the
□□□□□□*□terminal□and□v_state□is□the□a□bit□field□where□each□of□the
□□□□□□*□lower□16□bits□correspond□to□16□possible□combinations□of
□□□□□□*□terminals.□□The□following□is□the□bit□layout□of□v_state:
□□□□□□*
□□□□□□*□□|-------------|-----------|-----------|---------|
□□□□□□*□□|□15□14□13□12□|□11□10□9□8□|□7□6□5□6□4□|□3□2□1□0□|
□□□□□□*□□|-------------|-----------|-----------|---------|
□□□□□□*□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□^□^
□□□□□□*□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|□|
□□□□□□*□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|□--VT□0
□□□□□□*□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□|
□□□□□□*□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□--□VT□1
□□□□□□*/
□□□□□printf("The□active□vt□is□%x\n",□vtinfo.v_active);
□□□□□printf("The□signal□state□is□%x\n",□vtinfo.v_signal);
□□□□□printf("The□vt□state□is□%x\n",□vtinfo.v_state);
□□□□□□/*□Now□let's□print□out□the□state□of□all□the□Virtual□Terminals.
□□□□□□□*□This□can□be□accomplished□by□looking□at□the□v_state□and
□□□□□□□*□v_active□structure□elements.□*/
□□□□□for□(cnt□=□0;□cnt□<□15;□cnt++)□{
□□□□□□□□□□□□printf("\t/dev/vt%.2d□is□",□cnt);
□□□□□□□□□□□□if□(cnt□==□vtinfo.v_active)
□□□□□□□□□□□□□□□□□□printf("active.\n");
□□□□□□□□□□□□else□if□(vtinfo.v_state□&□(1□<<□cnt))
□□□□□□□□□□□□□□□□□□printf("open.\n");
□□□□□□□□□□□□else
□□□□□□□□□□□□□□□□□□printf("not□in□use.\n");
□□□□□}
□□□□□exit(0);
}
```

3.5.6.7 Virtual Terminal control

In Section "Writing well-behaved programs", the commands VT_GETMODE, VT_SETMODE and VT_RELDISP were used to synchronize a program with the end-user's VT switching activity. In this section we use the same commands to coordinate VT usage at the application-controlled level.

VT_GETMODE and VT_SETMODE are used to enable/disable the way the switching of the VTs takes place. The default action is to have the driver perform the switching function upon receipt of the appropriate keyboard sequence.

VT_GETMODE returns a structure of type "vt_mode". This structure defines the current VT switching state (VT_AUTO or VT_PROCESS) as well as signal and control information. Recall that VT_AUTO is the default mode and that VT_PROCESS allows you to take responsibility for managing the switching process and to take additional control of the switching mechanism.

The example provides two programs to demonstrate the functionality provided when a process chooses to manage VT switching.

The first program requests that a particular VT be activated. This is accomplished when the program executes the VT_ACTIVATE *ioctl*. VT_ACTIVATE informs *kd* that a program wishes to make a new VT be the active VT. This is the same as an end-user executing a hot-key sequence.

Before performing the switch, the driver sends a signal to the program that currently owns the active VT to request that it relinquish control. When the controlling VT receives the signal, it should execute the VT_RELDISP *ioctl* to inform the driver to proceed. Once this takes place, the driver performs the actual VT

switch.

This provides a mechanism for cooperation between the various programs controlling their VTs. Not only can a controlling display agree to relinquish the active VT slot, but it can also refuse (by using a 0 argument in the VT_RELDISP *ioctl*). In this example, the programs cooperate.

The test program is implemented by initiating two new virtual terminals manually (VT1 and VT2). Next, the 'acquire 1' command is invoked on VT1. Immediately (within 10 seconds), the user should manually switch to VT2 and execute the 'release' program. The reason for this contortion is once the release program is executed on VT2, the user's ability to make use of the automatic (VT_AUTO) VT switching capability is disabled. So, the release program starts first and sleeps to give the user time to manually switch VTs to VT2.

The following diagram illustrates the sequence of steps necessary to run the program.

VT1	User operation	VT2	Result
release 1			VT1 program sleeps 10 secs
	switch to VT2		VT2 displayed
		acquire	VT2 program waits for release request from VT1
signal sent to VT2 by "release" program			VT1 "acquire" program releases the active slot; Driver does the switch.
			VT1 is the actively displayed VT. VT2 program continues execution after handling the release signal. VT1 "release" terminates.

Table 28: Program flow with two Virtual Terminals

The result of the above efforts are that VT2 will be switched backed to VT1 as a result of program control instead of a manual keystroke.

Virtual Terminal control

```
□/*************************************************************************
□*□release□allows□the□Virtual□Terminal□to□be□manually
□*□released□under□program□control.□□It□is□used□in□conjunction
□*□with□'acquire'□to□demonstrate□the□VT_PROCESS□VT□mode.
□*************************************************************************
*/
#include□<stdio.h>
#include□<fcntl.h>
#include□<sys/types.h>
#include□<signal.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/vt.h>
#include□<errno.h>
int□□disp;
main()
{
□□□□□int□□□cnt,□i;
□□□□□int□release(),□acquire();
□□□□□struct□vt_stat□□□vtinfo;
□□□□□struct□vt_mode□□□vtmode;□□/*□structure□for□VT_GETMODE□*/
□□□□□/*□Open□the□controlling□terminal.□□If□successful,□the□device□is□an
□□□□□□*□appropriate□Virtual□Terminal.□*/
□□□□□if((disp□=□open("/dev/tty",□O_RDONLY))□<□0)□{
```

```
□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno□%d\n",
□□□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Get□the□video□mode□information.□□□*/
□□□□□if□(ioctl(disp,□VT_GETMODE,□&vtmode)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_GETMODE□failed:□errno□%d\n",
□□□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□printf("mode□is□%x\n",□vtmode.mode);□□□□□/*□VT_AUTO□or□VT_PROCESS□*
□□□□□printf("waitv□is□%x\n",□vtmode.waitv);□□□/*□Hang□on□writes□*
□□□□□printf("relsig□is□%x\n",□vtmode.relsig);□/*□release□signal□type□*
□□□□□printf("acqsig□is□%x\n",□vtmode.acqsig);□/*□acquire□signal□type□*
□□□□□printf("frsig□is□%x\n",□vtmode.frsig);□□□/*□forced□signal□type□*
□□□□□□/*□Now□that□we□have□acquired□the□vt_mode□structure,
□□□□□□□*□let's□change□the□way□we□switch□VTs.
□□□□□□□*□Instead□of□having□it□done□automatically□by□the□driver,
□□□□□□□*□we□can□request□to□do□this□within□the□application□program.
□□□□□□□*□The□release□and□acquire□signals□are□by□default□set
□□□□□□□*□to□SIGUSR1□and□SIGUSR2;□we□will□not□change□them.□*/
□□□□□vtmode.mode□=□VT_PROCESS;□□□/*□select□application□process□control□*/
□□□□□if□(ioctl(disp,□VT_SETMODE,□&vtmode)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_SETMODE□failed:□errno□%d\n",□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Set□up□to□catch□SIGUSR1□and□SIGUSR2□*/
□□□□□signal(SIGUSR1,□release);
□□□□□signal(SIGUSR2,□acquire);
□□□□□□□/*□For□simplicity,□assume□that□we□are□on□vt02□and□this□is□the active
□□□□□□□*□session.□Furthermore□we□wish□to□now□handle□requests□from□other
□□□□□□□*□processes□that□want□to□activate□VTs□other□than□our□active session
□□□□□□□*□"vt02".
□□□□□□□*□The□release□happens□when□our□process□receives□a□SIGUSR1.□□When□this
□□□□□□□*□occurs,□we□know□that□some□other□process□has□made□a□request□to the
□□□□□□□*□driver□to□activate□a□VT.□□release()□will□get□control□and□do□an
□□□□□□□*□ioctl□to□the□driver□to□give□up□control.□The□driver□will□then switch
□□□□□□□*□VTs□*/
□□□□□while(1)□{□□/*□loop□forever□*/
□□□□□□□□□□□□sleep(5);□□/*□sleep□5□seconds□*/
□□□□□□□□□□□□fprintf(stderr,"Alive□message□#□%d\n",i++);
□□□□□}
}
release()
{
□□□□□signal(SIGUSR1,□release);
□□□□□if(ioctl(disp,□VT_RELDISP,□1)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_RELDISP□(1)□failed,□errno□=□%d\n",
□□□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
}
/*
□*□Provided□for□completeness;□however,□this□function□will□not□be
□*□called.□□It□is□used□in□combination□with□the□waitv□structure□element
□*□in□combination□with□VT_GETMODE□for□additional□process□handshaking.
□*/
acquire()
{
□□□□□signal(SIGUSR2,□acquire);
□□□□□if(ioctl(disp,□VT_RELDISP,□VT_ACKACQ)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_RELDISP(VT_ACKACQ)□failed,□errno□=□%d\n",
□□□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
```

```
□□□□□}
}
/*********************************************************************
□*□This□is□the□acquire□command.□□It□is□used□to□request□a□particular
□*□VT□be□made□active.□□Its□syntax□is
□*
□*□□□acquire□VT_NUMBER
□*********************************************************************
**/
#include□<stdio.h>
#include□<fcntl.h>
#include□<sys/types.h>
#include□<signal.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/vt.h>
#include□<errno.h>
int□□disp;
main(argc,□argv)
int□□argc;
char□*argv[];
{
□□□□□int□□□□□□□□□□cnt,□nuvt,□i;
□□□□□struct□vt_stat□□□vtinfo;
□□□□□if(argc□!=□2){
□□□□□□□□□□□□fprintf(stderr,□"usage:□%s□vt_num\n",□argv[0]);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□nuvt□=□atoi(argv[1]);
□□□□□if((nuvt□<□1)□||□(nuvt□>□16))□{
□□□□□□□□□□□□fprintf(stderr,□"usage:□%s□vt_num\n",□argv[0]);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□/*□Wait□for□the□process□on□the□other□VT□to□be□manually□started.□*/
□□□□□sleep(10);
□□□□□/*□Open□/dev/video.□□*/
□□□□□if((disp□=□open("/dev/tty",□O_RDONLY))□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"Cannot□open□/dev/video,□errno□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□if□(ioctl(disp,□VT_ACTIVATE,□nuvt)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_ACTIVATE□failed:□errno□%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□exit(1);
□□□□□}
□□□□□sleep(5);□□□/*□sleep□5□seconds□*/
}
```

3.5.7 Miscellaneous capabilities

3.5.7.1 Setting borders

An application may set an EGA or VGA monitor to one of 63 different border colors by using the KDSBORDER *ioctl*. Bits 0 - 5 of the argument correspond to the colors blue, green, red, secondary blue, secondary green, and secondary red. These colors may be combined to generate different shades. An argument of 0 will turn off the border.

3.5.7.2 Keyboard operations

The KDGKBTYPE *ioctl* can be used to determine the type of keyboard attached to the system. The return value of this *ioctl* specifies whether an 84 key, 101 key, or unknown keyboard is attached.

The application can set and read the current "Num Lock", "Caps Lock" and "Scroll Lock" LED settings on the keyboard by using the KDSETLED and KDGETLED ioctls. Note that setting the "Num Lock" and "Caps Lock"

LEDs via the KDSETLED *ioctl* will have exactly the same effect as if you had depressed those keys manually. That is not the case when setting the "Scroll Lock" LED via software. Doing so will not suspend output to the screen as might be expected.

3.5.7.3 Sound effects

Use the KIOCSOUND and KDMKTONE *ioctl*s to add sound effects to an application. KIOCSOUND generates the same tone until called again either with a new argument for a new tone or a zero argument to turn off the tone altogether. KDMKTONE is similar except that the argument specifies the tone frequency as well as its duration.

3.5.7.4 Font operations

Applications can change the displayed text font. The default font information is stored on a ROM and does not consume system memory. Modifications to the font, therefore, cannot be overlaid but rather require use of kernel memory resources. For this reason, changing the font information is a capability that should be used only when necessary since storing a new font consumes additional system resources. A well-behaved application will restore the font information to its original state before exiting.

There are two different font programming interfaces. Both change the font not only for the active Virtual Terminal but for the other VTs as well. The first interface replaces the entire font, while the second interface allows particular characters within a font to be modified. In both cases, access is through an open file descriptor to */dev/video*. Use of one interface to change the font information undoes the changes made by use of the other interface, so the two interfaces cannot be used together.

Replacing the entire font

The first interface consists of the *ioctl*s *PIO_FONT8x8*, *PIO_FONT8x14*, *PIO_FONT8x16*, *GIO_FONT8x8*, *GIO_FONT8x14* and *GIO_FONT8x16*. The *GIO_ ioctl*s are used to obtain the current font information from the *KD* driver, and the *PIO_ ioctl*s are used to download the new font information to the video adapter.

Each of the *GIO_* and *PIO_ ioctl* system call requests are specific to a particular character box size (8x8, 8x14 and 8x16). The character box size is related to the current CGA/EGA/VGA text mode. To display a character in an 8x14 font requires 14 bytes of information. Each byte corresponds to one horizontal line of the font. Each bit within the byte corresponds to a pixel on that line, and the value of the bit is the pixel's on/off state. There are 256 characters per font, so an 8x14 font requires 14*256 = 3,584 bytes of storage. Similar logic applies to 8x16 and 8x8 fonts.

For both the *PIO_* and *GIO_ ioctl*s, the *arg* should be a pointer to an array of unsigned characters, the size of which is dependent on the character box size. The font information obtained by the *GIO_ ioctl*s or downloaded by the *PIO_ ioctl*s is of the same format as is used by the *vidi(1M)* command.

If a NULL pointer is supplied as the argument to the *PIO_ ioctl*s, the font is reset to the default system font.

Replacing characters within a font

The second interface consists of one *ioctl*: *WS_PIO_ROMFONT*. It is used to change the font information for any number of characters but does not require replacing the entire font. Rather, the changes are overlaid on top of the font information in the ROM. This interface is also different from the interface above in that changing the font information for a character cannot be done for just one character box size. Instead, the font information is supplied for each character box size: 8x8, 8x14, 8x16 and 9x16 (essentially the same as 8x16). The argument of the *ioctl* is a pointer to a *rom_font_t* structure (defined in */usr/include/sys/kd.h*). This structure contains the number of font entries being changed and the font information for each entry. An argument value of NULL restores the font information to the ROM font.

3.5.7.5 Programming the mouse

Most graphics applications assume that the end-user is using a mouse to move through the screen. The Reliant UNIX 5.43 Mouse Driver Package provides the system and command-level support for the operation of three types of mice. Information about how to install and configure support for mice is documented in the *System Administrator's Guide*, and programming information beyond what is discussed here is available in the description of *mouse(7)* in [7].

The application programmer's access to the mouse is the same regardless of the type of mouse (although a particular mouse may have special commands available to it). For most applications, interacting with a mouse is simply the process of:

- opening the mouse
- receiving mouse inputs for motion and button presses
- updating the screen to reflect the change in mouse state (including the display of the mouse cursor on the screen)
- repeating the previous two steps until the application is ready to release access to the mouse
- closing the mouse

The mouse is opened by opening the special file */dev/mouse*. If an error value is returned, there are three possible reasons:

1. The mouse is not attached or is not working.
2. The mouse has not been configured in the system and assigned a display terminal (see *mouseadmin(1M)*).
3. The controlling TTY of the process opening the mouse is not a virtual terminal to which a mouse has been assigned.

As was the case with the special file */dev/video*, access to */dev/mouse* requires that the controlling TTY of the process be a Virtual Terminal.

The system call *ioctl(2)* is used to obtain the current mouse state, using the *ioctl* command MOUSEIOCREAD. The system fills in a *mouseinfo* structure with the current mouse status information. The value of MOUSEIOCREAD and the definition of the *mouseinfo* structure are both in the file */usr/include/sys/mouse.h*, which should be included by applications using the mouse.

The *read(2)* and *write(2)* system calls are meaningless on the file */dev/mouse*.

The *mouseinfo* structure is defined as follows:

```
struct mouseinfo {
     unsigned char    status;
     char  xmotion, ymotion;
};
```

status contains the information about the current button state. Bit 0 (least significant) is 1 if mouse button 3 is depressed. Bits 1 and 2 similarly relate the button state of buttons 2 and 1, respectively. The *xmotion* and *ymotion* members reflect the change in movement that occurred since the last %MOUSEIOCTREAD *ioctl*, not the absolute x,y coordinates. The units of motion are 200 per inch. It is the program's responsibility to scale the change in mouse movement to a visual change in the mouse cursor's location on the screen. Larger scales require less mouse motion to traverse the screen but reduce the granularity of pointing.

In a Virtual Terminal environment, it is important that the application cease using MOUSEIOCREAD to process mouse events while its Virtual Terminal is not active. Otherwise, mouse events are potentially "stolen" from an application that is running in the active Virtual Terminal.

The following application uses the MOUSEIOCREAD *ioctl* to track mouse movement and also prints the current button status. It also uses the VT_SETMODE *ioctl* to put the Virtual Terminal in process mode, and demonstrates how to appropriately share the mouse in a virtual terminal environment. Screen control is done by using *libcurses* routines (see [15] for a description of *libcurses*):

```
#include <fcntl.h>
#include <sys/kd.h>
#include <sys/vt.h>
#include <errno.h>
#include <signal.h>
#include <curses.h>
#include <sys/mouse.h>
extern int errno;
int xscale = 10;
int yscale = 10;
int disp;  /* video file descriptor */
```

```
int mouse_is_on = 0; /* should we MOUSEIOCREAD or not? */
cleanup() {
    endwin();
    exit();
}
/*
 * VT release signal handler. Turn mouse off as part of releasing VT
 */
sigusr1()
{
    if (ioctl(disp, VT_RELDISP, 1) == -1) {
        fprintf(stderr, "VT_RELDISP failed, errno %d\n",
                errno);
        return(0);
    }
    mouse_is_on = 0;
    sigrelse(SIGUSR1);
}
/*
 * VT acquire signal handler. Turn mouse on as part of acquiring VT
 */
sigusr2()
{
    if (ioctl(disp, VT_RELDISP, VT_ACKACQ) == -1) {
        fprintf(stderr, "VT_RELDISP ack failed, errno %d\n",
                errno);
        return(0);
    }
    mouse_is_on = 1;
    sigrelse(SIGUSR2);
}
/*
 * invoke as mtracki <number>.
 * Mouse motions and button presses will be tracked.
 */
main(ac, av)
int ac;
char *av[];
{
    int msefd, x, y, sx, sy, old_sx, old_sy, sleep_time;
    struct mouseinfo m;
    struct vt_mode vtmode;
    if (ac != 2) {
        fprintf (stderr, "Usage: %s <sleep_time>\n", av[0])
        exit (1);
    }
    sleep_time = atoi (av[0]);
    if ((disp = open("/dev/video",O_RDWR)) < 0) {
        fprintf (stderr, "%s: can't open /dev/video; errno = %d\n"
                 av[0], errno);
        exit (1);
    }
    /* set signal handlers for VT process mode */
    signal (SIGINT, cleanup);
    signal (SIGUSR1, sigusr1);
    signal (SIGUSR2, sigusr2);
    /* set up for VT_SETMODE ioctl */
    vtmode.mode = VT_PROCESS;
    vtmode.relsig = SIGUSR1;
    vtmode.acqsig = SIGUSR2;
    vtmode.frsig = SIGUSR1; /* treat forced release same as release */
    vtmode.waitv = 0;
    errno = 0;
    /* go into process mode */
```

```
□□□□□if□(ioctl(disp,□VT_SETMODE,□&vtmode)□<□0)□{
□□□□□□□□□□□□fprintf(stderr,□"VT_SETMODE□failed:□errno□%d\n",□errno)
□□□□□□□□□□□□exit(1);
□□□□□}
/*□open□the□mouse□*/
□□□□□□if□((msefd□=□open□("/dev/mouse",□O_RDONLY))□<□0)□{
□□□□□□□□□□□□fprintf□(stderr,□"%s:□can't□open□/dev/mouse;□errno□=□%d\n"
□□□□□□□□□□□□□□□□□□□□av[0],□errno);
□□□□□□□□□□□□exit□(1);
□□□□□}
□□□□□mouse_is_on□=□1;
□□□□□/*□initialize□screen□output□using□curses(3x)□routines□*/
□□□□□initscr□();
□□□□□mvaddstr(LINES□-□1,□0,□"Mouse□tracking□with□ioctl's");
□□□□□refresh();
□□□□□/*□set□scale□and□initialize□mouse□positions□*/
□□□□□old_sx□=□sx□=□old_sy□=□sy□=□0;
□□□□□x□=□COLS□/□2□*□xscale;
□□□□□y□=□LINES□/□2□*□yscale;
□□□□□/*□loop□doing□MOUSEIOCREAD□ioctl.□VT_WAITACTIVE□ioctl□will
□□□□□□*□cause□the□process□to□sleep□until□its□VT□becomes□active
□□□□□□*□again.□Whether□the□VT□is□active□or□not□is□controlled□by
□□□□□□*□mouse_is_on
□□□□□□*/
□□□□□while□(1)□{
□□□□□□□□□□□□if□(sleep_time□>□0)
□□□□□□□□□□□□□□□□□□sleep(sleep_time);
□□□□□□□□□□□□if□(!mouse_is_on□&&□(ioctl(disp,VT_WAITACTIVE,0)□<□0))□{
□□□□□□□□□□□□□□□□□□fprintf□(stderr,□"%s:□can't□VT_WAITACTIVE;□errno□=\□%d\n"
□□□□□□□□□□□□□□□□□□□□□□□□av[0],□errno);
□□□□□□□□□□□□□□□□□□refresh□();
□□□□□□□□□□□□□□□□□□exit□(1);
□□□□□□□□□□□□}
□□□□□□□□□□□□if□(ioctl□(msefd,□MOUSEIOCREAD,□&m)□==□-1)□{
□□□□□□□□□□□□□□□□□□fprintf□(stderr,□"can't□ioctl;□errno□=□%d\n",
□□□□□□□□□□□□□□□□□□□□□□□□av[0],□errno);
□□□□□□□□□□□□□□□□□□refresh□();
□□□□□□□□□□□□□□□□□□exit□(1);
□□□□□□□□□□□□}
□□□□□□□□□□□□/*□update□mouse□cursor□position□*/
□□□□□□□□□□□□x□+=□m.xmotion;
□□□□□□□□□□□□y□+=□m.ymotion;
□□□□□□□□□□□□/*□erase□current□cursor□*/
□□□□□□□□□□□□mvaddch□(old_sy,□old_sx,□(int)□'□');
□□□□□□□□□□□□/*□compute□new□x,y□location□*/
□□□□□□□□□□□□if□((sx□=□x□/□xscale)□<□0)
□□□□□□□□□□□□□□□□□□x□=□sx□=□0;
□□□□□□□□□□□□else□if□(sx□>=□COLS)
□□□□□□□□□□□□□□□□□□x□=□(sx□=□COLS□-□1)□*□xscale;
□□□□□□□□□□□□if□((sy□=□y□/□yscale)□<□0)
□□□□□□□□□□□□□□□□□□y□=□sy□=□0;
□□□□□□□□□□□□else□if□(sy□>=□LINES□-□1)
□□□□□□□□□□□□□□□□□□y□=□(sy□=□LINES□-□2)□*□yscale;
□□□□□□□□□□□□/*□draw□new□mouse□cursor□*/
□□□□□□□□□□□□mvaddch□(sy,□sx,□(int)□'M');
□□□□□□□□□□□□old_sy□=□sy;
□□□□□□□□□□□□old_sx□=□sx;
□□□□□□□□□□□□/*□display□button□status.□Defines□are□in□mouse.h□*/
□□□□□□□□□□□□mvprintw□(0,□0,□"Status:□%02X\n",□m.status)
□□□□□□□□□□□□printw□("Buttons:□□1□%s□□2□%s□□3□%s",
□□□□□□□□□□□□□□□□□□m.status□&□BUT1STAT□?□"DN"□:□"UP",
□□□□□□□□□□□□□□□□□□m.status□&□BUT2STAT□?□"DN"□:□"UP",
□□□□□□□□□□□□□□□□□□m.status□&□BUT3STAT□?□"DN"□:□"UP");
□□□□□□□□□□□□/*□beep□if□the□button□state□changed□*/
```

```
□□□□□□□□□□□□if□(m.status□&□BUTCHNGMASK)
□□□□□□□□□□□□□□□□□□□□beep□();
□□□□□□□□□□□□refresh□();
□□□□□}
}
```

3.5.8 Comprehensive video programming example

```
------------------------□main.c□-------------------------------
#include□<stdio.h>
#include□<fcntl.h>
#include□<sys/types.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/signal.h>
#include□"vutil.h"
extern□FILE□□□□□□□□□□□□□□□□□□□□□□□□*logfp;□□□□□□□□□□/*□logfile□pointer□*/
extern□char□□□□□□□□□□□□□□□□□□□□□□□□*disptypes[];
extern□struct□kd_disparam□□□□□□□parms;
extern□struct□kd_memloc□□□□□□□□□□map;
extern□char□□□□□□□□□□□□□□□□□□□□□□□□scrmem[];
extern□char□□□□□□□□□□□□□□□□□□□□□□□□*screen;
extern□int□□□□□□□□□□□□□□□□□□□□□□□□disp;
extern□int□□□□□□□□□□□□□□□□□□□□□□□□errno;
extern□int□□□□□□□□□□□□□□□□□□□□□□□□save_mode;
main(argc,argv)
int□argc;
char□*argv[];
{
□□□□□□□□void□□□□□□□□□□□□□□□□□□□□□□*signal();
□□□□□□□□void□□□□□□□□□□□□□□□□□□□□□□sigtrap();
□□□□□□□□int□□□□□□□□□□□□□□□□□□□□□□□i;
□□□□□□□□int□□□□□□□□□□□□□□□□□□□□□□□indata;
□□□□□□□□extern□long□int□□□□□□□□□□end;
□□□□□□□□int□fd;
□□□□□/*□Open□the□kd□driver□□□□□□□□□□□□□□□□□□□□□□□□*/
□□□□□□□□if(open_driver()□!=□SUCCESS)□{
□□□□□□□□□□□□□□□□□□vreset();
□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□}
□□□□□□□□/*□Catch□program□termination□signals□and□clean□up□*/
□□□□□□□□signal(SIGHUP,sigtrap);□□□□□□□□□□/*□01□hang□up□*/
□□□□□□□□signal(SIGINT,sigtrap);□□□□□□□□□□/*□02□interrupt□*/
□□□□□□□□signal(SIGBUS,sigtrap);□□□□□□□□□□/*□10□bus□error□*/
□□□□□□□□signal(SIGSEGV,sigtrap);□□□□□□□□□/*□11□seg□violation□*/
□□□□□□□□/*□Set□the□video□mode□to□CGA□320x200□*/
□□□□□□□□if(set_video_mode(disp,□SW_CG320)□!=□SUCCESS)□{
□□□□□□□□□□□□□□□□□□vreset();
□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□}
□□□□□□□□/*□Retrieve□the□current□display□type,□video□memory
□□□□□□□□□□□□address,etc□*/
□□□□□□□□if(get_display_info(&parms)□!=□SUCCESS)□{
□□□□□□□□□□□□□□□□□□vreset();
□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□}
□□□□□□□□if(print_display_info(&parms)□!=□SUCCESS)□{
□□□□□□□□□□□□□□□□□□vreset();
□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□}
□□□□□□□□/*□Map□the□display□area□into□user□address□space.□*/
□□□□□□□□if(map_video_screen(CGA_SCREEN_SIZE,□1)□!=□SUCCESS)□{
□□□□□□□□□□□□□□□□□□vreset();
□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□}
```

```
□□□□□□□□/*□Paint□the□entire□screen□with□given□color□*/
□□□□□□□□loadmem(screen,(unsigned□char)□0x00,CGA_SCREEN_SIZE);□/*black*/
□□□□□□□□sleep(2);□□□□□□□/*□let□the□user□see□it□*/
□□□□□□□□loadmem(screen,(unsigned□char)□0x55,CGA_SCREEN_SIZE);□/*cyan*/
□□□□□□□□sleep(2);
□□□□□□□□loadmem(screen,(unsigned□char)□0xaa,CGA_SCREEN_SIZE);\□/*magenta*/
□□□□□□□□sleep(2);
□□□□□□□□loadmem(screen,(unsigned□char)□0xff,CGA_SCREEN_SIZE);□/*white*/
□□□□□□□□sleep(2);
□□□□□□□□clearmem(screen,CGA_SCREEN_SIZE);□□□□□□□/*□clear□the□display□*/
□□□□□□□□line(0,□0,□100,□100,□0x2);□□□□□□□□□□□□□□□/*□draw□a□line□*/
□□□□□□□□box(120,□120,□190,□190,□0x2);□□□□□□□□□□□/*□draw□a□box□*/
□□□□□□□□shade_box(120,□120,□190,□190,□0x2);□□□□□/*□fill□it□in□*/
□□□□□□□□circle(70,□220,□50,□0x01);□□□□□□□□□□□□□/*□draw□a□circle□*/
□□□□□□□□shade_circle(70,□220,□50,□0x01);□□□□□□□□/*□fill□it□in□*/
□□□□□□□□/*□Dump□the□screen□buffer□to□the□file□"memdump"□□□*/
□□□□□□□□disp_dump(screen,□CGA_SCREEN_SIZE);
□□□□□□□□sleep(2);
□□□□□□/*□Note□that□it□is□ESSENTIAL□to□unmap□and□reset□the□display\□prior□to
□□□□□□□*□exiting□the□program.□□Otherwise□the□display□is□left□in□an
□□□□□□□*□unusable□state.□*/
□□□□□□□□vreset();
}
/*-------------------------□util.c□-------------------------------*/
#include□<stdio.h>
#include□<fcntl.h>
#include□<sys/types.h>
#include□<sys/at_ansi.h>
#include□<sys/kd.h>
#include□<sys/signal.h>
#define□VGA_SCREEN_SIZE□□□□□□□□□□(1024□*□64)
#define□EGA_SCREEN_SIZE□□□□□□□□□□(1024□*□32)
#define□CGA_SCREEN_SIZE□□□□□□□□□□(1024□*□16)
#define□SUCCESS□□□□□□□□□□1
#define□FAIL□□□□□□□□□□□□□□0
#define□YES□□□□□□□□□□□□□□1
#define□NO□□□□□□□□□□□□□□□0
int□save_mode;
struct□kd_disparam□□□□□□parms;
char□□□□□□□□□□□□□□□□□□□□□□□□*screen,□*Tscreen;
int□□□□□□□□□□□□□□□□□□□□□□□□Vinit□=□NO;□□□□□□□□□/*□initialization□flag□*/
int□□□□□□□□□□□□□□□□□□□□□□□□map_flag□=□0;
int□□□□□□□□□□□□□□□□□□□□□□□□disp;□□□□□□□□□□□□□□□/*□console□file□descriptor□*/
FILE□□□□□□□□□□□□□□□□□□□□□□□□*logfp;
extern□□□□□□□□□□□□□□□□□□□□□□□□errno;
log(format,arg)
char□*format;
int□arg;
{
□□□□□□□□fprintf(logfp,format,arg);
□□□□□□□□fflush(logfp);
□□□□□□□□return(SUCCESS);
}
/*□sigtrap()□is□to□protect□against□a□signal□terminating□the□process
□*□and□leaving□the□video□display□in□an□inconsistent□state.
□*/
void□sigtrap(sig)
int□sig;
{
□□□□□□□□log("Trapped□signal□=□%x,□exiting...\n",sig);
□□□□□□□□vreset();
□□□□□□□□fclose(logfp);
□□□□□□□□exit(0);
}
```

```
vreset()
{
□□□□□□□□□if□(map_flag)□{
□□□□□□□□□□□□□□□□□if(unmap_video_screen()□!=□SUCCESS)□{
□□□□□□□□□□□□□□□□□□□□□□□□□□log("ERROR:□Unmap_video_screen()□failed\n");
□□□□□□□□□□□□□□□□□□□□□□□□□□vreset();
□□□□□□□□□□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□□□map_flag□=□0;
□□□□□□□□□}
□□□□□□□□□if□(save_mode)□□/*□Reset□to□original□display□mode:□*/
□□□□□□□□□□□□□□□□□if□(ioctl(disp,□MODESWITCH□|□save_mode,□0)□<□0)□{
□□□□□□□□□□□□□□□□□□□□□□□□□□log("ERROR:□Unable□to□reset,□mode:□%x\n",
□□□□□□□□□□□□□□□□□□□□□□□□□save_mode);
□□□□□□□□□□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□□□□□□□□□}
□□□□□□□□□else□/*□Reset□to□default□text□mode:□*/
□□□□□□□□□□□□□□□□□if□(ioctl(disp,□KDSETMODE,□KD_TEXT)□<□0)□{
□□□□□□□□□□□□□□□□□□□□□□□□□□log("Unable□to□reset□display□to□text□mode\n");
□□□□□□□□□□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□□□□□□□□□}
□□□□□□□□□return(SUCCESS);
}
/*
□*□This□subroutine□will□open□the□log□file□and□console□driver□and□will save
□*□the□current□display□mode□settings□for□later□reset.□□It□must□be
□*□executed□before□attempting□any□access□of□the□video□driver□functions.
□*/
open_driver()
{
□□□□□□□□□Vinit□=□1;□□□□□□□/*□set□the□initialization□flag□*/
□□□□□□□□□if((logfp□=□fopen("video_log","w+"))□<□0)□{
□□□□□□□□□□□□□□□□□log("ERROR:□could□not□open□logfile\n");
□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□}
□□□□□□□□□log("Opened□logfile\n");
□□□□□□□□□if((disp□=□open("/dev/console",□(O_RDWR□|□O_NDELAY)))□<□0)□{
□□□□□□□□□□□□□□□□□log("ERROR:□open(dev/console)□failed,□errno□=□$%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□}
□□□□□□□□□if□((save_mode□=□ioctl(disp,□CONS_GET,□0))□<□0)□{
□□□□□□□□□□□□□□□□□log("CONS_GET□failed\n");
□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□}
□□□□□□□□□return(SUCCESS);
}
/*
□*□This□routine□will□retrieve□display□parameters□from□the□video□driver.
□*□Specifically,□the□display□type,□video□memory□address□and□valid□I/O
□*□addresses.
□*/
get_display_info(kdp)
struct□kd_disparam□*kdp;
{/*
□□□□□□□□□□*□ioctl(KDDISPTYPE)□will□return□a□structure□populated□with
□□□□□□□□□□*□the□type□of□display,□the□physical□memory□location□of□the
□□□□□□□□□□*□screen□memory□and□valid□I/O□port□addresses.
□□□□□□□□□□*
□□□□□□□□□□*□struct□kd_disparam□{
□□□□□□□□□□*□□□□□□□long□type;
□□□□□□□□□□*□□□□□□□char□*addr;
□□□□□□□□□□*□□□□□□□ushort□ioaddr[MKDIOADDR];
□□□□□□□□□□*□};
```

```
□□□□□□□□□□*/
□□□□□□□□□if(ioctl(disp,□KDDISPTYPE,□kdp)□<□0)□{
□□□□□□□□□□□□□□□□□□log("ERROR:□ioctl(KDDISPTYPE)□failed,□errno□=□$%d\n",
□□□□□□□□□□□□□□□□□□errno);
□□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□}
□□□□□□□□□return(SUCCESS);
}
/*□This□function□will□print□the□kd_disparam□structure□contents□to□the□logfile.
□*□Call□get_display_info()□first.
□*/
print_display_info(kdp)
struct□kd_disparam□*kdp;
{
□□□□□□□□□int□i;
□□□□□□□□□char□*type;
□□□□□□□□□switch□(kdp->type)□{
□□□□□□□□□□□□□□□□□case□KD_MONO:□□□□□□□□□□□□□type="MONOCHROME";□□break;
□□□□□□□□□□□□□□□□□case□KD_HERCULES:□□□□□□□□type="HERCULES";□□□□break;
□□□□□□□□□□□□□□□□□case□KD_CGA:□□□□□□□□□□□□□□type="CGA";□□□□□□□□□□break;
□□□□□□□□□□□□□□□□□case□KD_EGA:□□□□□□□□□□□□□□type="EGA";□□□□□□□□□□break;
□□□□□□□□□□□□□□□□□case□KD_VGA:□□□□□□□□□□□□□□type="VGA";□□□□□□□□□□break;
□□□□□□□□□□□□□□□□□default:□□□□□□□□□□□□□□□□□□type="Invalid□type";break;
□□□□□□□□□}
□□□□□□□□□log("display□type□□=□%s\n",□type);
□□□□□□□□□log("Video□Address□=□0x%lx\n",□kdp->addr);
□□□□□□□□□for□(i□=□0;□i□<□MKDBASEIO;□i++)□{
□□□□□□□□□□□□□□□□□□log("0x%.3x\n",kdp->ioaddr[i]);
□□□□□□□□□}
□□□□□□□□□return(SUCCESS);
}
/*set_video_mode()will□accept□a□passed□argument□and□set□the□video□mode
□*□accordingly.
□*/
set_video_mode(fd,mode)
int□fd;
int□mode;
{
□□□□□□□□□if(ioctl(fd,□mode)□<□0)□{
□□□□□□□□□□□□□log("ERROR:□set_video_mode:□ioctl(%x)□failed,□errno□=□$%d\n",
□□□□□□□□□□□□□mode,□errno);
□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□}
□□□□□□□□□return(SUCCESS);
}
/*
□*□map_video_screen()□will□map□the□video□memory□into□thee□user's□address
□*□space.The□arguments□to□this□function□are:
□*□Length□of□memory□to□map,□□□□□-□16K□for□CGA□mode□5
□*□I/O□address□enable□flag.□□□□□-□Needed□to□do□inp/outp
□*/
map_video_screen(length,ioflg)
long□length;
long□ioflg;
{
□□□□□□□□□struct□kd_disparam□□□□□□□kd_param;
□□□□□□□□□struct□kd_memloc□□□□□□□□□map;
□□□□□□□□□Tscreen□=□(char□*)malloc(length□+□4096);
□□□□□□□□□screen□=□(char□*)((unsigned)(Tscreen□+□4095)□&□~4095);
□□□□□□□□□if□(ioctl(disp,□KDDISPTYPE,□&kd_param)□==□-1)□{
□□□□□□□□□□□□□□□□□□log("KDDISPTYPE□failed,□errno:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□□}
□□□□□□□□□map.physaddr□=□kd_param.addr;□□□/*□set□the□video□address□*/
```

```
        map.vaddr = screen;                /* set the virtual address */
        map.length = length;               /* set the screen size */
        map.ioflg = ioflg;                 /* enable i/o addresses */
        if(ioctl(disp, KDMAPDISP, &map) < 0) {
                log("ERROR: KDMAPDISP failed, errno = %d\n",errno);
                return(FAIL);
        }
        map_flag = 1;
        return(SUCCESS);
}
/*
 * unmap_video_screen() will release the mapped memory.  This should
 * be called before exit by the program that called map_video_screen().
 * It releases the video screen so that other programs can map it.
 */
unmap_video_screen()
{
        if (map_flag) {
                if(ioctl(disp, KDUNMAPDISP) < 0) {
                        log("ERROR: ioctl(KDUNMAPDISP) failed, errno = %d\n",
                                errno);
                        return(FAIL);
                }
        } else {
                log("ERROR: unmap_video_screen: display not mapped.\n");
                return(FAIL);
        }
        map_flag = 0;
        return(SUCCESS);
}
/*
 * disp_dump will dump the contents of the video screen to disk in a file
 * called 'memdump'. Useful for debugging.
 */
disp_dump(addr,dsize)
char *addr;
int dsize;
{
        int fd;
        if((fd = open("memdump",O_CREAT | O_WRONLY, 0644)) < 0) {
                log("ERROR: open(memdump) failed, errno = %d\n",errno);
                return(FAIL);
        }
        if(write(fd, addr, dsize) < 0) {
                log("ERROR: write(memdump) failed, errno = %d\n",errno);
                return(FAIL);
        }
        close(fd);
}
/*
 * loadmem() will clear the screen to a selected color attribute based on
 * the passed in variable color.
 */
loadmem(ptr,color,count)
char *ptr;
unsigned char color;
int count;
{
        int i;
        for(i = 0; i<count; i++) {
                *ptr++ = color;
        }
        return(SUCCESS);
}
```

```
/*
□*□This□routine□will□write□one□screen□point□in□320x200□color
□*□mode.□□The□bit□layout□for□video□memory□in□this□mode□is
□*
□*□□□□□□□byte
□*□□---------------
□*□□7□6□5□4□3□2□1□0
□*
□*
□*□□□□□□pixel
□*□□□---------------
□*□□□00□□11□□22□□33□□□□□<-□byte□0
□*□□□44□□55□□66□□77□□□□□<-□byte□1
□*
□*□□□Start□Address□=□0xB8000□for□even□rows
□*
*□□□Start□Address□=□0xB8000□+□8192□□=□BA000□for□odd□rows
□*
□*
□*□Each□byte□of□screen□memory□holds□4□screen□points□worth□of□data.□ There
□*□are□2□bits□per□screen□pixel□to□allow□color□representation□(00,01,10,11).
□*□The□video□memory□is□segmented□into□segments□where□the□even□rows
□*□start□at□address□0xB8000□and□the□odd□rows□at□0xB8000□+□8192.□ Therefore
□*□to□write□the□first□4□pixels□on□the□screen,□it□would□be□necessary□to
□*□write□the□first□byte□at□0xB8000.□□To□write□to□the□next□4□points□on the
□*□first□line,□the□first□byte□at□0xB8000□+□1□would□have□to□be□written.
□*□To□write□the□first□4□pixels□on□the□2nd□screen□line,□the□address□to
□*□be□written□must□be□(0xB8000□+□8192□=□0xBA000).
□*
□*
□*□This□function□is□specific□to□CGA□but□can□be□modified□to□work□with
□*□EGA□and□VGA.
□*/
unsigned□int□bmask[4]□=□{0xff3f,□0xffcf,□0xfff3,□0xfffc};
int□color_shift[4]□=□{6,□4,□2,□0};
int□write_pixel(row,col,color)
int□row,□col,□color;
{
□□□□int□□□□□index;□□□□□□□□□□□/*□screen□byte□index□*/
□□□□unsigned□int□mask;
□□□□char□*sptr;□□□□□□□□□□□□□/*□screen□memory□pointer□*/
□□□□sptr□=□screen;□□□□□□□□□□/*□set□pointer□to□screen□area□*/
□□□□color□<<=□color_shift[col□%□4];
□□□□mask□=□bmask[col□%□4];
□□□□/*
□□□□□*□Find□the□correct□position□in□the□screen□memory.
□□□□□*□The□video□memory□is□set□up□with□odd/even□rows□residing□at different
□□□□□*□(non)contiguous□memory□addresses.
□□□□□*□Because□of□this,□we□need□to□ensure□that□the□starting□address□of the
□□□□□*□byte□we□need□to□access□is□indeed□correct.
□□□□□*/
□□□□□index□=□row□*□40□+□col/4;
□□□□□/*
□□□□□□*□Now□decide□whether□the□row□was□even□or□odd.□□If□the□row
□□□□□□*□is□odd,□use□(B8000□+□index□+□8152).□□If□the□row
□□□□□□*□is□even,□use□(B8000□+□index).□□This□means
□□□□□□*□that□the□first□odd□row□starts□at□(B8000□+□40□+□8152□=□BA000)
□□□□□□*/
□□□□□□□□if(row□%□2)□index□+=□8152;□□□□□□/*□bank□2□if□odd□*/
□□□□□□*(sptr□+□index)□=□color□|□*(sptr□+□index)□&□mask;
□□□□□□return(SUCCESS);
}
/*
□*□This□function□will□draw□a□line□in□a□specified□color.
```

```
□*/
line(start_row,□start_col,□end_row,□end_col,□color)
int□□□□□start_row,
□□□□□□□□□start_col,
□□□□□□□□□end_row,
□□□□□□□□□end_col,
□□□□□□□□□color;
{
□□□□□□□□□register□int□□□□i,
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□length;
□□□□□□□□□int□□□□□□□□□□□□□□□□ydiff,
□□□□□□□□□□□□□□□□□□□□□□□□□□□□xdiff,
□□□□□□□□□□□□□□□□□□□□□□□□□□□□inc_row,
□□□□□□□□□□□□□□□□□□□□□□□□□□□□inc_col;
□□□□□/*
□□□□□□*□Determine□which□way□the□line□is□sloping□and□the□appropriate
□□□□□□*□directional□increment.
□□□□□□*□Now□determine□the□row□and□column□increment□value
□□□□□□*/
□□□□□□□□if((ydiff□=□end_row□-□start_row)□>□0)
□□□□□□□□□□□□□□□□□inc_row□=□1;
□□□□□□□□else□if(ydiff□==□0)
□□□□□□□□□□□□□□□□□inc_row□=□0;
□□□□□□□□else
□□□□□□□□□□□□□□□□□inc_row□=□-1;
□□□□□□□□if((xdiff□=□end_col□-□start_col)□>□0)
□□□□□□□□□□□□□□□□□inc_col□=□1;
□□□□□□□□else□if(xdiff□==□0)
□□□□□□□□□□□□□□□□□inc_col□=□0;
□□□□□□□□else
□□□□□□□□□□□□□□□□□inc_col□=□-1;
□□□□□□□□/*
□□□□□□□□□*□Determine□which□length□is□greater
□□□□□□□□□*/
□□□□□□□□if(abs(ydiff)□>□abs(xdiff))
□□□□□□□□□□□□□□□□□length□=□abs(ydiff);
□□□□□□□□else
□□□□□□□□□□□□□□□□□length□=□abs(xdiff);
□□□□□□□□/*
□□□□□□□□□*□Now□draw□the□line.
□□□□□□□□□*/
□□□□□□□□for(i□=□0;□i<=□length;□i++)□{
□□□□□□□□□□□□□□□□□write_pixel(start_row,□start_col,□color);
□□□□□□□□□□□□□□□□□start_row□+=□inc_row;
□□□□□□□□□□□□□□□□□start_col□+=□inc_col;
□□□□□□□□}
□□□□□□□□return(SUCCESS);
}
/*
□*□This□function□will□draw□a□box.□□It□makes□use□of□line()□which
□*□then□makes□use□of□write_pixel().
□*/
box(start_row,□start_col,□end_row,□end_col,□color)
int□□□□□start_row,
□□□□□□□□□start_col,
□□□□□□□□□end_row,
□□□□□□□□□end_col,
□□□□□□□□□color;
{
□□□□□□□□if(line(start_row,□start_col,□end_row,□start_col,color)□!=□SUCCESS)
□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□if(line(start_row,□start_col,□start_row,□end_col,color)□!=□SUCCESS)
□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□if(line(start_row,□end_col,□end_row,□end_col,color)□!=□SUCCESS)
```

```
□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□if(line(end_row,□start_col,□end_row,□end_col,color)□!=□SUCCESS)
□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□return(SUCCESS);
}
/*
□*□This□function□will□shade□a□box.□□This□is□another□function□being provided
□*□to□demonstrate□the□graphics□capabilities□that□could□be□implemented
□*□with□this□interface.□□It□works□by□using□line()□to□shade□in□the
□*□previously□created□box.
□*/
shade_box(start_row,□start_col,□end_row,□end_col,□color)
int□□□□□start_row,
□□□□□□□□start_col,
□□□□□□□□end_row,
□□□□□□□□end_col,
□□□□□□□□color;
{
□□□□□□□□register□int□□□□i,
□□□□□□□□□□□□□□□□□□□□□□□□□□□start,
□□□□□□□□□□□□□□□□□□□□□□□□□□□end;
□□□□□if(start_row□<□end_row)□{
□□□□□□□□□□□□□start□=□start_row;
□□□□□□□□□□□□□end□=□end_row;
□□□□□}
□□□□□else□{
□□□□□□□□□□□□□start□=□end_row;
□□□□□□□□□□□□□end□=□start_row;
□□□□□}
□□□□□□□□for(i□=□start;□i□<=□end;□i++)□{
□□□□□□□□□□□□□□□□□if(line(i,□start_col,□i,□end_col,□color)□!=□SUCCESS)
□□□□□□□□□□□□□□□□□□□□□□□□□□□□return(FAIL);
□□□□□□□□}
□□□□□□□□return(SUCCESS);
}
/*
□*□This□routine□will□draw□a□circle
□*/
circle(xlocus,□ylocus,□radius,□color)
int□□xlocus,
□□□□□ylocus,
□□□□□radius,
□□□□□color;
{
□□□□□register□int□x,□diff;
□□□□□diff□=□radius/2;
□□□□□for(x□=□0;□x□<□radius;□x++)□{
□□□□□□□□write_pixel(radius□+□xlocus,□x□+□ylocus,□color);
□□□□□□□□write_pixel(xlocus□-□radius,□x□+□ylocus,□color);
□□□□□□□□write_pixel(radius□+□xlocus,□ylocus□-□x,□color);
□□□□□□□□write_pixel(xlocus□-□radius,□ylocus□-□x,□color);
□□□□□□□□write_pixel(x□+□xlocus,□radius□+□ylocus,□color);
□□□□□□□□write_pixel(xlocus□-□x,□radius□+□ylocus,□color);
□□□□□□□□write_pixel(x□+□xlocus,□ylocus□-□radius,□color);
□□□□□□□□write_pixel(xlocus□-□x,□ylocus□-□radius,□color);
□□□□□□□□□□□if(diff□<□0)
□□□□□□□□□□□□□□□□diff□+=□□radius--□-□x;
□□□□□□□□□□else
□□□□□□□□□□□□□□□□diff□-=□x;
□□□□□}
□□□□□if(radius)□{
□□□□□□□□write_pixel(radius□+□xlocus,□x□+□ylocus,□color);
□□□□□□□□write_pixel(xlocus□-□radius,□x□+□ylocus,□color);
□□□□□□□□write_pixel(radius□+□xlocus,□ylocus□-□x,□color);
```

```
□□□□□□□write_pixel(xlocus□-□radius,□ylocus□-□x,□color);
□□□□□□□write_pixel(x□+□xlocus,□radius□+□ylocus,□color);
□□□□□□□write_pixel(xlocus□-□x,□radius□+□ylocus,□color);
□□□□□□□write_pixel(x□+□xlocus,□ylocus□-□radius,□color);
□□□□□□□write_pixel(xlocus□-□x,□ylocus□-□radius,□color);
□□□□□}
□□□□□return(SUCCESS);
}
/*
□*□this□will□fill□in□a□circle□by□calling□circle()□repeatedly
□*□with□smaller□circle□values.
□*/
shade_circle(a,□b,□c,□d)
int□□□□□a,□b,□c,□d;
{
□□□□□while(c--)
□□□□□□□□□□circle(a,b,c,d);
□□□□□return(SUCCESS);
}
/*-----------------------□vtutil.h□-------------------------------*/
#define□VGA_SCREEN_SIZE□□□□□□□□□□(1024□*□64)
#define□EGA_SCREEN_SIZE□□□□□□□□□□(1024□*□32)
#define□CGA_SCREEN_SIZE□□□□□□□□□□(1024□*□16)
#define□SUCCESS□□□□□□□□□□1
#define□FAIL□□□□□□□□□□□□□0
#define□YES□□□□□□□□□□□□□□1
#define□NO□□□□□□□□□□□□□□□0
```

3.5.9 Graphics mode summary descriptions

3.5.9.1 SW_BG320

Description:
320x200 Black & White Graphics Mode

Mode:
CGA Mode 4

Memory requirements:
8K per page (2 pages)

Map 0:
B8000 - B9F3F

Pixel layout:
One bit per pixel

3.5.9.2 SW_CG320

Description:
320x200, 4 colors

Mode:
CGA Mode 5

Memory map requirements:
6K per page (2 pages)

Map 0:
B8000, B8002, ... B9F3E (even scans); BA000, BA002, ... BBF3E (odd scans)

Map 1:
B8001, B8003, ... B9F3F (even scans); BA001, BA003, ... BBF3F (odd scans)

Pixel layout:
2 bits per pixel as follows:

```
---------------------------------
|□□P0□□|□□□P1□□|□□□P2□□|□□□P3□□|
```

```
--------------------------------
□7□□□6□□□5□□□4□□□3□□□2□□□1□□□0
```

Pixel byte mapping alternates between Map 0 and Map 1. Byte B8000 contains first 4 pixels in upper left hand corner of display. Byte B8001 contains the next 4 in that first row, etc. Addresses B8000 - B9F3F map all the pixels in the even scan lines, while addresses BA000 - BBF3F map all the pixels in the odd scan lines.

Color selection:

0 0 : Black

0 1 : Light Cyan

1 0 : Light Magenta

1 1 : Intense White

3.5.9.3 SW_BG640

Description:
640x200, 2 colors

Mode:
CGA Mode 6

Memory map requirements:
6K per page (2 pages)

This mode has the same mapping and addressing scheme as SW_CG320 above, except the data format layout is 1 bit per pixel as follows:

```
----------------------------------------
|□P0□|□P1□|□P2□|□P3□|□P4□|□P5□|□P6□|□P7□|
----------------------------------------
□□7□□□□6□□□□5□□□□4□□□□3□□□□2□□□□1□□□□0
```

Color selection:

0 : Black

1 : Intense White

3.5.9.4 SW_CG320_D

Description:
320 x 200, 16 colors

Mode:
EGA Mode D

Memory requirements:
8K per page (8 pages)

Map 0:
A0000 - A1F3F, blue bit plane (C0)

Map 1:
A0000 - A1F3F, green bit plane (C1)

Map 2:
A0000 - A1F3F, red bit plane (C2)

Map 3:
A0000 - A1F3F, intensity bit plane (C3)

Pixel layout:
4 bits per pixel as follows:

C3:
P0[3]|P1[3]|P2[3]|P3[3]|P4[3]|P5[3]|P6[3]|P7[3]|

C2:

P0[2]|P1[2]|P2[2]|P3[2]|P4[2]|P5[2]|P6[2]|P7[2]|

C1:

P0[1]|P1[1]|P2[1]|P3[1]|P4[1]|P5[1]|P6[1]|P7[1]|

C0:

P0[0]|P1[0]|P2[0]|P3[0]|P4[0]|P5[0]|P6[0]|P7[0]|

7 6 5 4 3 2 1 0

Each of the 4 maps provides one bit of a pixel's color.

Color selection:

C3	C2	C1	C0	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	White
1	0	0	0	Dark Gray
1	0	0	1	Light Blue
1	0	1	0	Light Green
1	0	1	1	Light Cyan
1	1	0	0	Light Red
1	1	0	1	Light Magenta
1	1	1	0	Yellow
1	1	1	1	Intense White

3.5.9.5 SW_CG640_E

Description:

640 x 200, 16 colors

Mode:

EGA Mode E

Memory requirements:

16K per page (4 pages)

Map 0:

A0000 - A3E7F, blue bit plane (C0)

Map 1:

A0000 - A3E7F, green bit plane (C0)

Map 2:

A0000 - A3E7F, red bit plane (C0)

Map 3:

A0000 - A3E7F, intensity bit plane (C0)

Pixel layout and color selection are identical to that of SW_CG320_D.

3.5.9.6 SW_CG640x350

Description:
640 x 350, 4 colors (Valid only for EGA systems with 64K video RAM)

Mode:
EGA Mode 10

Memory requirements:
32K per page (2 pages)

Map 0:
A0000 - A6D5F, blue bit plane (C0)

Map 1:
A0000 - A6D5F, green bit plane (C1)

Map 2:
A0000 - A6D5F, red bit plane (C2)

Map 3:
A0000 - A6D5F, intensity bit plane (C3)

Pixel layout and color selection are identical to that of SW_CG320_D except that Maps 0 and 2 are chained together to provide a 4 bit color code for pixels at even addresses, and Maps 1 and 3 are chained together to provide 4 bit color codes for pixels at odd addresses.

3.5.9.7 SW_ENH_CG640

Description:
640 x 350 16 colors (Valid only for EGA systems with 128K video RAM)

Mode:
EGA Mode 10*

Memory requirements:
128K

Map 0:
A0000 - A6D5F, blue bit plane (C0)

Map 1:
A0000 - A6D5F, green blue bit plane (C1)

Map 2:
A0000 - A6D5F, red bit plane (C2)

Map 3:
A0000 - A6D5F, intensity bit plane (C3)

Pixel layout and color selection are identical to that of SW_CG320_D.

This *ioctl(2)* system call is the same as SW_CG640x350, except it is used for systems configured with a minimum of 128K bytes of video memory.

3.5.9.8 SW_VGA640x480C

Description:
640 x 480, 2 colors

Mode:
VGA Mode 11

Memory requirements:
64K per page (1 page)

Map 0:
A0000 - A95FF

Pixel layout:

|□P0□|□P1□|□P2□|□P3□|□P4□|□P5□|□P6□|□P7|

□□7□□□□6□□□□5□□□□4□□□□3□□□□2□□□□1□□□□0

Color selection:

0 : Black

1 : Intense White

3.5.9.9 SW_VGA640x480E

Description:

640 x 480, 16 colors from 256K

Mode:

VGA Mode 12

Memory requirements:

64K - 1 page

Map 0:

A0000 - A95FF, blue bit plane (C0)

Map 1:

A0000 - A95FF, green bit plane (C1)

Map 2:

A0000 - A95FF, red bit plane (C2)

Map 3:

A0000 - A95FF, intensity bit plane (C3)

Pixel layout and color selection are identical to that of SW_CG320_D.

3.5.9.10 SW_VGA320x200

Description:

320x200, 256 colors

Mode:

VGA Mode 13

Memory requirements:

64K - 1 page

Map 0:

A0000, A0004, A0008, ... AF9FC

Map 1:

A0001, A0005, A0009, ... AF9FD

Map 2:

A0002, A0006, A000A, ... AF9FE

Map 3:

A0003, A0007, A000B, ... AF9FF

Pixel layout: 8 bits per pixel (1 pixel per byte)

Color selection:

8 bits select one out of a possible 256 color registers. Each color register has 3 components, corresponding to a value for RED, GREEN and BLUE. Each component is represented by 6 bits:

□□□□□□□6□Bits□□□□□6□Bits□□□□□6□Bits

|□□□□□red□□□□|□□□green□□|□□□blue□|□□□□□□□□□□Color□Register

3.5.9.11 SW_ATT640

Description:
AT&T Enhancement - 640 x 400, 16 colors

Mode:
AT&T Enhancement

Memory requirements:
64K

Map 0:
A0000 - A7DFF, blue bit plane (C0)

Map 1:
A0000 - A7DFF, blue bit plane (C1)

Map 2:
A0000 - A7DFF, blue bit plane (C2)

Map 3:
A0000 - A7DFF, blue bit plane (C3)

Pixel layout and color layout identical to SW_C6320_D.

3.5.9.12SW_VDC800x600E

Description:
AT&T enhancement- 800 x 600, 16 colors from 256K

Mode:
EGA Mode 12

Memory requirements:
64K

Map 0:
A0000 - AEA5F, blue bit plane (C0)

Map 1:
A0000 - AEA5F, green bit plane (C1)

Map 2:
A0000 - AEA5F, red bit plane (C2)

Map 3:
A0000 - AEA5F, intensity bit plane (C3)

Pixel layout and color selection are identical to that of SW_C6320_D.

3.5.9.13SW_VDC640x400V

Description:
AT&T Enhancement - 640 x 400, 256 colors from 256K

Mode:
AT&T enhancement

Memory requirements:
64K

Map 0:
A0000 - AF9FF(1st Quadrant)

Map 1:
A0000 - AF9FF(2nd Quadrant)

Map 2:
A0000 - AF9FF(3rd Quadrant)

Map 3:
A0000 - AF9FF(4th Quadrant)

Pixel layout and color selection are identical to that of SW_VGA320x200

3.5.10 Text and graphics mode ioctls

3.5.10.1 Text mode selection ioctls

Description	Note	Adapter	IOCTL
40x25 B&W		VGA, EGA & CGA	SW_B40x25
40x25 Color	CGA Mode 1		SW_C40x25
80x25 B&W			SW_C80x25
80x25 Color	CGA Mode 3	SW_C80x25	
40x25 B&W		VGA & EGA	SW_ENHB40x25
40x25 Color	EGA Mode 0,1		SW_ENHC40x25
80x25 B&W			SW_ENHB80x25
80x25 Color	EGA Mode 2,3		SW_ENHC80x25
80x25 Mono	EGA Mode 7		SW_EGAMONO80x25
80x43 B&W		EGA only	SW_ENHB80x43
80x43 Color		EGA only	SW_ENHB80x43
40x25 Color	VGA Mode 0,1	VGA only	SW_VGAC40x25
80x25 Color	VGA Mode 2,3		SW_VGAC80x25
80x25 Mono	VGA Mode 7		SW_VGAMONO80x25

Table 29: Text mode selection ioctls

3.5.10.2 Graphics mode selection IOCTLs

Description	Note	Adapter	IOCTL
320x200 B&W	CGA Mode 4	VGA, EGA & CGA	SW_BG320
320x200 4 Color	CGA Mode 5		SW_CG320
640x200 B&W	CGA Mode 6		SW_BG640
320x200 16 Color	EGA Mode D	VGA & EGA	SW_CG320_D
640x200 16 Color	EGA Mode E		SW_CG640_E
640x350 Mono	EGA Mode F		SW_EGAMONOAPA
640x350 Mono	EGA Mode F*		SW_ENH_MONOAPA2
640x350 4 Color	EGA Mode 10		SW_CG640x350
640x350 16 Color	EGA Mode 10*		SW_ENH_CG640
640x480 2 Color	VGA Mode 11	VGA only	SW_VGA640x480C
640x480 16 Color	VGA Mode 12		SW_VGA640x480E
320x200 256	VGA Mode		SW_VGA320x200

Color	13		
640x400 16 Color		AT&T VDC 750, 600	SW_ATT640
800x600 16 Color		AT&T VDC 600	SW_VDC800x600E
600x400 256 Color		AT&T VDC 600	SW_VDC640x400V

Table 30: Graphics mode selection ioctls

3.5.10.3display(7) ioctl summary

Ioctl	**Description**	**Arguments**	**Return value**
*KIOCINFO	Identifies driver	none	if KD driver, returns (('k'<<8)|'d')
KDDISPTYPE	Display info	(struct □ □□kd_disparam *)□ □□□□□□□□ □arg	struct kd_disparam {□ □□long type;□ □□char *addr;□ □□ushort ioaddr[MKDIOADDR];□ } Valid values for type field: □ KD_MONO□ KD_HERCULES□ KD_CGA□ KD_EGA□ KD_VGA
KDVDCTYPE	Adapter info	(struct □ □□kd_vdctype *)□ □□□□□□□□ □arg	struct kd_vdctype {□ □□long cntlr;□ □□long disply;□ □□long rsrvd;□ } Valid Valid Cntrlr Values Display Values □ KD_MONO KD_UNKNOWN□ KD_HERCULES KD_STAND_M□ KD_CGA KD_STAND_C□ KD_EGA KD_MULTI_M□ KD_VGA KD_MULTI_C□ KD_VDC400□ KD_VDC750□ KD_VDC600

KDGKBTYPE	Keyboard type	(char *)□ □□□□□□□□ □arg	KD_84□□□/*84 Key Keyboard*/□ KD_101 /*101 Key Keyboard*/□ KB_OTHER
KDGETMODE	Display mode	(int *)□ □□□□□□□□ □arg	KD_TEXT /*clear screen*/□ KD_TEXT1□□□□□/*don't clear*/□ KD_GRAPHICS /*Graphics mode*/
KDSETMODE	Set mode	(int) arg	
CONS_GET	Get mode	none	decode for specific mode,e.g. SW_CG320, SW_ENHC80x25,...
GIO_ATTR	Get attributes	none	decode for FG and BG color
KDSBORDER	Set border	(char) arg	
KDMAPDISP	Maps memory	(struct□□□□ □□kd_memloc *)□ □□□□□□□□ □arg	struct kd_memloc {□ □□□char *vaddr; /*map TO*/□ □□□char *physaddr; /*FROM*/□ □□□long length; /* # to map*/□ □□□long ioflg; /*enable I/O*/□ }
KDUNMAPDISP	Unmap	none	
KDENABIO	Enable Video □ I/O	none	
KDDISABIO	Disable Video □ I/O	none	
KDADDIO	Add I/O port	(unsigned short)□ □□□□□□□□ □arg	
KDDELIO	Delete I/O port	(unsigned short)□ □□□□□□□□ □arg	
KDQUEMODE	Enable/disablequeue	(struct □ □□kd_quemode *)□ □□□□□□□□ □arg	struct kd_quemode {□ □□□int qsize; /* # in q*/□ □□□int signo;□□/*sig to send*/□

			□□□char *qaddr; /*vaddr of q*/□ }
KDGETLED	Get LED status	(char *)□ □□□□□□□□ □arg	LED_SCR□ LED_CAP□ LED_NUM
KDSETLED	Set LED status	(char *)□ □□□□□□□□ □arg	
KIOCSOUND	Generate sound	(int)□□arg	
KDMKTONE	Generate tone	(int)□□arg	
*VT_OPENQRY	Find VT	(long) arg	first available VT #
VT_GETMODE	Get VT mode	(struct□□□ □□□vt_mode *)□ □□□□□□□ar g	struct vt_mode {□ □□char Modus;□ □□char waitv;□ □□short relsig;□ □□short acqsig;□ □□short frsig;□ } mode field values: □ VT_AUTO□□□□□□□/*a uto switch*/□ VT_PROCESS /*process switch*/
VT_SETMODE	Set VT mode	(struct□□□□□ □□□ □□□vt_mode *) □ □□□□□□□ar g	
VT_RELDISP	Release status	(int)□□arg	
VT_ACTIVATE	Make VT active	(int)□□arg	
VT_WAITACTIV E	Wait untilVT active	none	

Table 31: Text mode selection ioctls

Entries marked by " * " are not applicable when the application opens */dev/video*. For these ioctls, use a file descriptor to the Virtual Terminal itself.

3.6 liber, A Library System

To illustrate the use of Reliant UNIX system programming tools in the development of an application, we are going to pretend we are engaged in the development of a computer system for a library. The system is known as *liber*. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop

short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficacy of this design. It is the way it is only in order to provide some passably realistic examples of Reliant UNIX system programming tools in use. It is not an application, but rather is code fragments only.

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

The design of the system calls for it to be brought up at the beginning of the day and remain running while the library is in operation. Associated with each terminal is a program specific to the function of that terminal, each running as a separate Reliant UNIX process. The system has one master index that contains the unique identifier of each title in the library. When the system is running the index is mapped into the address space of each process. Semaphores are used to synchronize access to the index. In the pages that follow fragments of some of the system's programs are shown to illustrate the way they work together. The startup program performs the system initialization; opening the semaphores and the index file; mapping the index file into memory; and kicking off the other programs. The id numbers for the semaphores (*wrtsem* and *rdsem*) are written to a file during initialization, this file is then read by all the subsidiary programs so that all use the same semaphores.

The following code allows all the programs to share access to the index file provided that a new process is then spawned for the other programs (*fork*):

```
/*
□*□Gain□access□to□the□index□file,□map□it□in.
□*□After□mapping,□free□the□file□descriptor□so
□*□that□it□will□be□available□for□other□uses□--
□*□the□mapping□will□remain□until□the□program
□*□exits,□or□until□the□mapping□is□removed□either
□*□by□munmap()□or□by□mapping□over□top□of□this□one
□*□with□another□call□to□mmap().□□Note□the□use□of
□*□the□read/write□open□mode□--□all□programs□but
□*□"add-books"□should□open□just□for□read-only.
□*/
if□((index_fd□=□open("index.file",□O_RDWR))□==□-1)
{
□□□□□□(void)□fprintf(stderr,□"index□open□failed:□%d\n",□errno)
□□□□□□exit(1);
}
/*
□*□Establish□the□mapping.□□As□with□the□call□to
□*□open(),□all□programs□but□"add-books"□should
□*□map□with□PROT_READ□for□read-only□access.
□*/
if□((int)(index□=□(INDEX□*)mmap(0,□sizeof□(INDEX),□PROT_READ|PROT_WRITE,
□□□□MAP_SHARED,□index_fd,□0)□==□-1)
{
□□□□□□(void)□fprintf(stderr,□"mmap□failed:□%d\n",□errno)
□□□□□□exit(1);
}
(void)□close(index_fd);
```

The preceding code fragment establishes a mapping to the index file in the address space of the program. Access to the addresses at which the file is mapped affect the file directly, no further file operations are required. For instance, if the access deposits data at the accessed address, then the file will be modified by operation. If the access examines data, then the file will be accessed. In either case, the portion of the file containing the information will be obtained or restored to secondary storage automatically by the system and transparently to the *liber* application.

Of the programs shown, *add-books* is the only one that alters the index. The semaphores are used to ensure

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

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

```
/*□liber.h□-□header□file□for□the
□□□□□□□□□□□□□□□□□□□□□□□□*□□□□□□□□□□□□library□system.
□□□□□□□□□□□□□□□□□□□□□□□□*/
typedef□...□INDEX;□□□□□□□□□/*□data□structure□for□book□file□index□*/
typedef□struct□{□□□□□□□□□□□/*□type□of□records□in□book□file□*/
□□□□□□char□title[30];
□□□□□□char□author[30];
□□□□□□.
□□□□□□.
□□□□□□.
}□BOOK;
int□index_fd;
int□wrtsem;
int□rdsem;
INDEX□*index;
int□book_file;
BOOK□book_buf;
/*□□□□startup□program□□□□□*/
/*□1.□Open□index□file□and□map□it□in.
□*□2.□Open□two□semaphores□for□providing□exclusive□write□access□to□index.
□*□3.□Stash□id's□for□shared□memory□segment□and□semaphores□in□a□file
□*□□□□where□they□can□be□accessed□by□the□programs.
□*□4.□Start□programs:□□add-books,□card-catalog,□and□checkout□running
□*□□□□on□the□various□terminals□throughout□the□library.
□*/
#include□□□□□<stdio.h>
#include□□□□□<sys/types.h>
#include□□□□□<sys/ipc.h>
#include□□□□□<sys/shm.h>
#include□□□□□<sys/sem.h>
#include□□□□□"liber.h"
void□exit();
extern□int□errno;
key_t□key;
int□shmid;
int□wrtsem;
int□rdsem;
FILE□*ipc_file;
main()
{
□□□□□□.
□□□□□□.
□□□□□□.
□□□□□□/*
□□□□□□□*□Open□index□file□and□map□it.
□□□□□□□*/
□□□□□□/*□See□previous□example□*/
□□□□□□/*
□□□□□□□*□Get□the□read/write□semaphores.
□□□□□□□*/
□□□□□□if□((wrtsem□=□semget(key,□1,□IPC_CREAT□|□0666))□==□-1)
□□□□□□{
□□□□□□□□□(void)□fprintf(stderr,□"startup:□semget□failed:□errno=%d\n",□errno);
□□□□□□□□□exit(1);
□□□□□□}
□□□□□□if□((rdsem□=□semget(key,□1,□IPC_CREAT□|□0666))□==□-1)
□□□□□□{
□□□□□□□□□(void)□fprintf(stderr,□"startup:□semget□failed:□errno=%d\n",□errno);
```

```
□□□□□□□□□□exit(1);
□□□□□□□}
□□□□□□□(void)□fprintf(ipc_file,□"%d\n%d\n",□wrtsem,□rdsem
□□□□□□□/*
□□□□□□□□*□fork□the□add-books□program□running□on□the□terminal□in□the
□□□□□□□□*□basement.□□Start□the□checkout□and□card-catalog□programs
□□□□□□□□*□running□on□the□various□other□terminals□throughout□the□library.
□□□□□□□□*/
□□□□□□□.
□□□□□□□.
□□□□□□□.
}
/*□□□□card-catalog□program*/
/*□1.□Read□screen□for□author□and□title.
□*□2.□Use□semaphores□to□prevent□reading□index□while□it□is□being□written.
□*□3.□Use□index□to□get□position□of□book□record□in□book□file.
□*□4.□Print□book□record□on□screen□or□indicate□book□was□not□found.
□*□5.□Go□to□1.
□*/
#include□□□□□□□□□<stdio.h>
#include□□□□□□□□□<sys/types.h>
#include□□□□□□□□□<sys/ipc.h>
#include□□□□□□□□□<sys/sem.h>
#include□□□□□□□□□<fcntl.h>
#include□□□□□□□□□"liber.h"
void□exit();
extern□int□errno;
struct□sembuf□sop[1];
main()□{
□□□□□□□.
□□□□□□□.
□□□□□□□.
□□□□□□□while□(1)□□□/*□loop□*/
□□□□□□□{
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Read□author/title/subject□information□from□screen.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Wait□for□write□semaphore□to□reach□0□(index□not□being
□□□□□□□□□□□□□□□*□written).
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□sop[0].sem_op□=□1;
□□□□□□□□□□□□□□□if□(semop(wrtsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Increment□read□semaphore□so□potential□writer□will□wait
□□□□□□□□□□□□□□□*□for□us□to□finish□reading□the□index.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□sop[0].sem_op□=□0;
□□□□□□□□□□□□□□□if□(semop(rdsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*□Use□index□to□find□file□pointer(s)□for□book(s)□*/
/*□Decrement□read□semaphore□*/
□□□□□□□□□□□□□□□sop[0].sem_op□=□-1;
□□□□□□□□□□□□□□□if□(semop(rdsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□□exit(1);
```

```
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□□*□Now□we□use□the□file□pointers□found□in□the□index□to
□□□□□□□□□□□□□□□□*□read□the□book□file.□□Then□we□print□the□information
□□□□□□□□□□□□□□□□*□on□the□book(s)□to□the□screen.
□□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□□*□Note□design□alternatives□for□this□portion□of□the
□□□□□□□□□□□□□□□□*□the□code:□the□book□file□could□be□accessed□by
□□□□□□□□□□□□□□□□*□lseek()s□to□the□portion□of□the□file□containing
□□□□□□□□□□□□□□□□*□the□record,□and□then□read()□could□be□used□to
□□□□□□□□□□□□□□□□*□obtain□the□file□information.□□Alternatively,□the
□□□□□□□□□□□□□□□□*□entire□book□file□could□be□mapped□into□memory,□and□the
□□□□□□□□□□□□□□□□*□the□record□accessed□directly□without□further
□□□□□□□□□□□□□□□□*□file□operations,□or□the□area□of□the□file□containing
□□□□□□□□□□□□□□□□*□the□book□record□could□just□be□mapped□and□then□unmapped
□□□□□□□□□□□□□□□□*□when□the□access□is□complete.
□□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□□.
□□□□□□□□□□□□□□□□.
□□□□□□□□□□□□□□□□.
□□□□□□□}□/*□while□*/
}
/*□□□□checkout□program□□□□*/
/*
□*□1.□Read□screen□for□Dewey□Decimal□number□of□book□to□be□checked□out.
□*□2.□Use□semaphores□to□prevent□reading□index□while□it□is□being□written.
□*□3.□Use□index□to□get□position□of□book□record□in□book□file.
□*□4.□If□book□not□found□print□message□on□screen,□otherwise□lock
□*□□□□book□record□and□read.
□*□5.□If□book□already□checked□out□print□message□on□screen,□otherwise
□*□□□□mark□record□"checked□out"□and□write□back□to□book□file.
□*□6.□Unlock□book□record.
□*□7.□Go□to□1.
□*/
#include□□□□□□□□□<stdio.h>
#include□□□□□□□□□<sys/types.h>
#include□□□□□□□□□<sys/ipc.h>
#include□□□□□□□□□<sys/sem.h>
#include□□□□□□□□□<fcntl.h>
#include□□□□□□□□□"liber.h"
void□exit();
long□lseek();
extern□int□errno;
struct□flock□flk;
struct□sembuf□sop[1];
long□bookpos;
main()
{
□□□□□□.
□□□□□□.
□□□□□□.
□□□□□□while□(1)
□□□□□□{
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□□*□Read□Dewey□Decimal□number□from□screen.
□□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□□*□Wait□for□write□semaphore□to□reach□0□(index□not□being
□□□□□□□□□□□□□□□□*□written).
□□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□sop[0].sem_flg□=□0;
□□□□□□□□□□□□□□□sop[0].sem_op□=□0;
□□□□□□□□□□□□□□□if□(semop(wrtsem,□sop,□1)□==□-1)
```

```
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Increment□read□semaphore□so□potential□writer□will□wait
□□□□□□□□□□□□□□□*□for□us□to□finish□reading□the□index.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□sop[0].sem_op□=□1;
□□□□□□□□□□□□□□□if□(semop(rdsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Now□we□can□use□the□index□to□find□the□book's□record\ position.
□□□□□□□□□□□□□□□*□Assign□this□value□to□"bookpos".
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□/*□Decrement□read□semaphore□*/
□□□□□□□□□□□□□□□sop[0].sem_op□=□-1;
□□□□□□□□□□□□□□□if□(semop(rdsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Lock□the□book's□record□in□book□file,□read□the□record.
□□□□□□□□□□□□□□□*□Here□again□we□have□the□design□option□of□deciding□to
□□□□□□□□□□□□□□□*□access□and□update□the□database□through□the□use□of
□□□□□□□□□□□□□□□*□seeks,□read()s□and□write()s;□or□file□mapping□can
□□□□□□□□□□□□□□□*□be□used□to□access□the□file.□□File□mapping□has□the
□□□□□□□□□□□□□□□*□disadvantage□that□it□does□not□interact□well□with
□□□□□□□□□□□□□□□*□enforcement-mode□locking,□although□semaphores
□□□□□□□□□□□□□□□*□could□be□used□as□an□alternative□synchronization
□□□□□□□□□□□□□□□*□mechanism□to□file□locking.□□File□mapping□would□have
□□□□□□□□□□□□□□□*□potential□efficiency□advantages,□eliminating□the□need
□□□□□□□□□□□□□□□*□for□repetitive□file□access□operations□and□attendant
□□□□□□□□□□□□□□□*□data□copying.□□For□this□example,□however,□we□choose
□□□□□□□□□□□□□□□*□not□to□use□mapping□to□demonstrate□the□use□of□other
□□□□□□□□□□□□□□□*□system□facilities.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□flk.l_type□=□F_WRLCK;
□□□□□□□□□□□□□□□flk.l_whence□=□0;
□□□□□□□□□□□□□□□flk.l_start□=□bookpos;
□□□□□□□□□□□□□□□flk.l_len□=□sizeof(BOOK);
□□□□□□□□□□□□□□□if□(fcntl(book_file,□F_SETLKW,□&flk)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"trouble□locking:□%d\n",\ errno);
□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□if□(lseek(book_file,□bookpos,□0)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□(Error□processing□for□lseek);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□if□(read(book_file,□&book_buf,□sizeof(BOOK))□==□-1)
{
□□□□□□□□□□□□□□□□□□□□□(Error□processing□for□read);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□If□the□book□is□checked□out□inform□the□client,□otherwise
□□□□□□□□□□□□□□□*□mark□the□book's□record□as□checked□out□and□write□it
□□□□□□□□□□□□□□□*□back□into□the□book□file.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□/*□Unlock□the□book's□record□in□book□file.□*/
```

```
□□□□□□□□□□□□□□□flk.l_type□=□F_UNLCK;
□□□□□□□□□□□□□□□if□(fcntl(book_file,□F_SETLK,□&flk)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"trouble□unlocking:□%d\n",\ errno);
□□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□}□/*□while□*/
}
/*□□□□add-books□program□□□*/
/*
□*□1.□Read□a□new□book□entry□from□screen.
□*□2.□Insert□book□in□book□file.
□*□3.□Use□semaphore□"wrtsem"□to□block□new□readers.
□*□4.□Wait□for□semaphore□"rdsem"□to□reach□0.
□*□5.□Insert□book□into□index.
□*□6.□Decrement□wrtsem.
□*□7.□Go□to□1.
□*/
#include□□□□□<stdio.h>
#include□□□□□<sys/types.h>
#include□□□□□<sys/ipc.h>
#include□□□□□<sys/sem.h>
#include□□□□□"liber.h"
void□exit();
extern□int□errno;
struct□sembuf□sop[1];
BOOK□bookbuf;
main()
{
□□□□□□.
□□□□□□.
□□□□□□.
□□□□□□for□(;;)
□□□□□□{
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Read□information□on□new□book□from□screen.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□addscr(&bookbuf);
□□□□□□□□□□□□□□□/*□write□new□record□at□the□end□of□the□bookfile.
□□□□□□□□□□□□□□□*□Code□not□shown,□but
□□□□□□□□□□□□□□□*□addscr()□returns□a□1□if□title□information□has
□□□□□□□□□□□□□□□*□been□entered,□0□if□not.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Increment□write□semaphore,□blocking□new□readers□from
□□□□□□□□□□□□□□□*□accessing□the□index.
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□sop[0].sem_flg□=□0;
□□□□□□□□□□□□□□□sop[0].sem_op□=□1;
□□□□□□□□□□□□□□□if□(semop(wrtsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
□□□□□□□□□□□□□□□/*
□□□□□□□□□□□□□□□*□Wait□for□read□semaphore□to□reach□0□(all□readers□to finish
□□□□□□□□□□□□□□□*□using□the□index).
□□□□□□□□□□□□□□□*/
□□□□□□□□□□□□□□□sop[0].sem_op□=□0;
□□□□□□□□□□□□□□□if□(semop(rdsem,□sop,□1)□==□-1)
□□□□□□□□□□□□□□□{
□□□□□□□□□□□□□□□□□□□□□□□(void)□fprintf(stderr,□"semop□failed:□%d\n",□errno);
□□□□□□□□□□□□□□□□□□□□□□□exit(1);
□□□□□□□□□□□□□□□}
```

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

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

1. Information read in from a *curses* window can be stored in fields that are part of a structure defined in the header file for the application.
2. The address of the structure can be passed from another function where the record is processed.

```
/*□□addscr□is□called□from□add-books.
□□□□□□□□□□□□□□□□□□□□□□□□□*□□The□user□is□prompted□for□title
□□□□□□□□□□□□□□□□□□□□□□□□□*□□information.
□□□□□□□□□□□□□□□□□□□□□□□□□*/
#include□<curses.h>
WINDOW□*cmdwin;
addscr(bb)
struct□BOOK□*bb;
{
□□□□□□int□c;
initscr();
□□□□□□nonl();
□□□□□□noecho();
□□□□□□cbreak();
□□□□□□cmdwin□=□newwin(6,□40,□3,□20);
□□□□□□mvprintw(0,□0,□"This□screen□is□for□adding□titles□to□the□data□base");
□□□□□□mvprintw(1,□0,□"Enter□□a□□to□add;□□q□□to□quit:□");
□□□□□□refresh();
□□□□□□for□(;;)
□□□□□□{
□□□□□□□□□□□□□□□refresh();
□□□□□□□□□□□□□□□c□=□getch();
□□□□□□□□□□□□□□□switch□(c)□{
□□□□□□□□□□□□□□□case□'a':
□□□□□□□□□□□□□□□□□□□□□□werase(cmdwin);
□□□□□□□□□□□□□□□□□□□□□□box(cmdwin,□'|',□'-');
□□□□□□□□□□□□□□□□□□□□□□mvwprintw(cmdwin,□1,□1,□"Enter□title:□");
□□□□□□□□□□□□□□□□□□□□□□wmove(cmdwin,□2,□1);
□□□□□□□□□□□□□□□□□□□□□□echo();
□□□□□□□□□□□□□□□□□□□□□□wrefresh(cmdwin);
□□□□□□□□□□□□□□□□□□□□□□wgetstr(cmdwin,□bb->title);
□□□□□□□□□□□□□□□□□□□□□□noecho();
□□□□□□□□□□□□□□□□□□□□□□werase(cmdwin);
□□□□□□□□□□□□□□□□□□□□□□box(cmdwin,□'|',□'-');
□□□□□□□□□□□□□□□□□□□□□□mvwprintw(cmdwin,□1,□1,□"Enter□author:□");
□□□□□□□□□□□□□□□□□□□□□□wmove(cmdwin,□2,□1);
□□□□□□□□□□□□□□□□□□□□□□echo();
□□□□□□□□□□□□□□□□□□□□□□wrefresh(cmdwin);
□□□□□□□□□□□□□□□□□□□□□□wgetstr(cmdwin,□bb->author);
□□□□□□□□□□□□□□□□□□□□□□noecho();
□□□□□□□□□□□□□□□□□□□□□□werase(cmdwin);
□□□□□□□□□□□□□□□□□□□□□□wrefresh(cmdwin);
```

```
□□□□□□□□□□□□□□□□□□□□□□endwin();
□□□□□□□□□□□□□□□□□□□□□□return(1);
□□□□□□□□□□□□□□□□case□'q':
□□□□□□□□□□□□□□□□□□□□□□erase();
□□□□□□□□□□□□□□□□□□□□□□endwin();
□□□□□□□□□□□□□□□□□□□□□□return(0);
□□□□□□□□□□□□□□□□}
□□□□□□}
}
#
#□Makefile□for□liber□library□system
#
CC□=□cc
CFLAGS□=□-O
all:□startup□add-books□checkout□card-catalog
startup:□liber.h□startup.c
□□□□□□$(CC)□$(CFLAGS)□-o□startup□startup.c
add-books:□add-books.o□addscr.o
□□□□□□$(CC)□$(CFLAGS)□-o□add-books□add-books.o□addscr.o
add-books.o:□liber.h
checkout:□liber.h□checkout.c
□□□□□□$(CC)□$(CFLAGS)□-o□checkout□checkout.c
card-catalog:□liber.h□card-catalog.c
□□□□□□$(CC)□$(CFLAGS)□-o□card-catalog□card-catalog.c
```

4 Packaging application and driver software

This chapter comprises the following descriptions:

- Packaging application software
- Modifying the *sysadm* interface
- Data validation tools
- Package installation case studies

4.1 Packaging application software

This section describes how to package software that will be installed on a computer running Reliant UNIX 5.43. The approach to packaging in a Reliant UNIX 5.43 environment differs from a pre-Version 5.4x environment. Pre-Version 5.4x packages deliver information to the system through script actions but a Reliant UNIX 5.43 package does this through package information files. A packaging tool, the *pkgmk* command, is provided to help automate package creation. It gathers the components of a package on the development machine, copies them onto the installation medium, and places them into a structure that *pkgadd* recognizes.

This chapter also describes the installation tool, the *pkgadd* command, which copies the package from the installation medium onto a system and performs system housekeeping routines that concern the package. This tool is primarily for the installer but is described here to provide you with a background on the environment into which your packages will be placed and to help you test-install your packages.

The next two sections describe what a package consists of and gives an overview of the structural life cycle of a package (how its structure on your development machine relates to its structure on the installation medium and on the installation machine).

The remaining sections familiarize you with all of the tools, files, and scripts involved in creating a package, provide suggestions for how to approach software packaging, and describe some specific procedures. After reading this chapter, you should study Section "Package installation case studies", which provides case studies using the tools and techniques described in the present section.

All of the commands, files, and functions mentioned in this section have manual entries in the Chapter "Reference pages".

4.1.1 Contents of a package

A software package is made up of a group of components that together create the software. These components naturally include the executables that comprise the software, but they also include at least two information files and can optionally include other information files and scripts.

As shown in the following figure, a package's contents fall into three categories:

1. required components (the *pkginfo* file, the *prototype* file, package objects)
2. optional package information files
3. optional packaging scripts

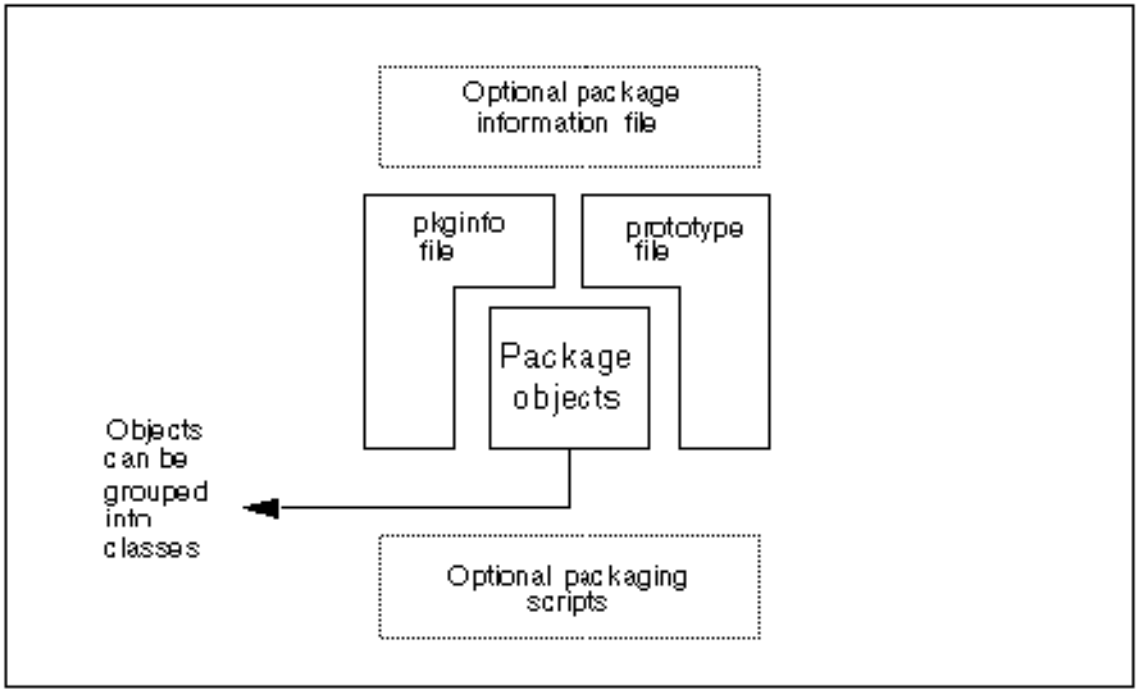


Figure 12: The contents of a package

4.1.1.1 Required components

At the very least, a package must contain the following components:

- package objects

  These are the objects that make up the software. They can be files (executable or data), directories, or named pipes. Objects can be manipulated in groups during installation by placing them into classes. You will learn more about classes when reading the section "Placing Objects into Classes."

- the *pkginfo* file

  The *pkginfo* file is a required package information file defining parameter values that describe a package. For example, this file defines values for the package abbreviation, the full package name, and the package architecture.

- the *prototype* file

  The *prototype* file is a required package information file that lists the contents of the package. There is one entry for each deliverable object and this entry consists of several fields of information describing the object. All package components, including the *pkginfo* file, must be listed in the *prototype* file.

Both required package information files are described further in "The package information files" section and on their respective manual entries in Chapter "Reference pages".

4.1.1.2 Optional package information files

There are four optional package information files that you can add to your package:

1. the *compver* file

   Defines previous versions of the package that are compatible with this version.

2. the *depend* file

   Defines any software dependencies associated with this package.

3. the *space* file

   Defines disk space requirements for the target environment beyond that used by objects defined in the *prototype* file (for example, files that will be dynamically created at installation time).

4. the *copyright* file

   Defines the text for a copyright message that will be printed on the terminal at the time of package installation or removal.

Every package information file used must have an entry in the *prototype* file. All of these files are described further in Section "The package information files", and in their respective manual entries in Chapter "Reference pages".

4.1.1.3 Optional installation scripts

Your package can use three types of installation scripts, although no scripts are required. Many of the tasks executed in a pre-Version 5.41 installation script are now accomplished automatically by *pkgadd*. However, you may deliver scripts with a Version 5.41 package to perform customized actions. An installation script must be executable by *sh* (for example, a shell script or executable program). The three script types are the request script (solicits system administrator input), class action script (defines a set of actions to perform on a group of objects), and the procedure script (defines actions that will occur at particular points during installation).

Packaging scripts are described in detail in "The installation scripts" section.

4.1.2 The structural life cycle of a package

The material covered in this chapter talks about package object pathnames. You should keep in mind while reading that a package object will reside in three places while being packaged and installed. To help you avoid confusion, consider which of the three possible locations are being discussed:

1. On a development machine

   Packages originate on a development machine. They can be in the same directory structure on your machine as they will be placed on the installation machine. Or *pkgmk* can locate components on the development machine and give them different pathnames on the installation machine.

2. On the installation media

   When *pkgmk* copies the package components from the development machine to the installation medium, it places them into the structure you have defined in your *prototype* file and a format that *pkgadd* recognizes.

3. On the installation machine

   pkgadd copies a package from the installation medium and places it in the structure defined in your *prototype* file. Package objects can be defined as relocatable, meaning the installer can define the actual location of these package objects on the installation machine during installation. Objects with fixed locations are copied to their predefined path.

4.1.3 The packaging tools

The packaging tools are provided to automate package creation and to remove the burden of packaging from the developer. There are three packaging tools:

1. *pkgmk* copies the components of a package from the development machine to a fixed directory structure and performs all necessary formatting. Optionally it may copy the package onto the installation medium.
2. *pkgtrans* translates an installable package from one package format to another. The two format types are directory structure and datastream. For example, *pkgmk* creates a directory structure. You would use *pkgtrans* to translate a package already formatted as a directory structure into a datastream format.
3. *pkgproto* generates a *prototype* file based on the directory structure of your development area.

If the *XS224_DEBUG* shell variable is set, additional information (about system calls, for example) is written to *stderr* at command runtime. The output begins with "**XS224_DEBUG:". The scope of the output can be extended at any time.

Any path names specified in a call to one of these commands must be absolute path names.

Each of these commands is described in the following text and has a manual entry in Chapter "Reference pages".

4.1.3.1 The pkgmk command

This command takes all of the package objects residing on the development machine, optionally copies them onto the installation medium, and places them into a fixed directory structure. You are not required to know the details of the fixed directory structure since *pkgmk* takes care of the formatting. However, for your information, the "Sample device drivers" section describes the two types of package formats supported by these tools: a fixed directory structure and a datastream structure.

Files can be unstructured on the development machine and *pkgmk* will structure them correctly on the medium based on information supplied in the *prototype* file. The installation medium onto which a package is formatted can be what is typically thought of as a medium (a diskette, for example) or it can be a directory on a machine. Exceptions include installation media such as SMC cartridges: for these you have to use *pkgtrans* instead of *pkgmk*.

pkgmk requires the presence of two information files on the development machine, the *prototype* and the *pkginfo* file (other package information files may be present). The *pkginfo* file defines the values for a number of package parameters, such as the package abbreviation and the package name. The *prototype* file provides a complete list of the package contents. *pkgmk* creates the *pkgmap* file, which is the package contents file on the *pkgmk*-formatted installation medium, by processing the *prototype* file and then adding three fields to each entry.

pkgmk follows these steps when processing a package:

- Processes all of the command lines in the input *prototype* file. (*prototype* command lines can tell *pkgmk* where to look for package objects, to merge another *prototype* into this one, define a default *mode owner group* set for package objects, and can place a parameter value in the packaging environment.)
- Copies the objects of a package onto the *pkgmk*-formatted installation medium, using the *prototype* file as a listing of contents.
- Puts the package objects into the proper format.
- Divides a package into pieces and distributes those pieces on multiple volumes, if necessary. Files larger than the capacity of a single volume cannot be handled.
- Creates the *pkgmap* file (the content listing file that is placed on the installation medium). It looks like the *prototype* file except that all command lines are processed, and the *volno*, *size*, *cksum*, and *modtime* fields are added to each entry.

4.1.3.2 The pkgtrans command

This command translates a package already created with *pkgmk* from one package format to another. It can make the following translations:

- a fixed file system structure to a datastream
- a datastream to a fixed file system structure
- a fixed file system structure to a fixed directory structure

Note that a package in a fixed directory structure can be in a directory on disk (for example, in a spooling directory) or on a removable device such as a diskette. A datastream can be on any device; for example, on a diskette or a tape.

4.1.3.3 The pkgproto command

This command generates a *prototype* file. It scans the paths specified on the command line and creates description line entries for these paths. If the pathname is a directory, an entry for each object in the directory is generated. You can use the *–c* option of the *pkgproto* command to place objects into classes.

When you create a *prototype* file with an editor, it does not matter how package components are organized on your development machine. You use the *path1=path2* pathname format to define where the files reside on your development machine and where they should be placed on the installation machine. However, when you use *pkgproto* to create your file, your development area must be structured exactly as you wish your package to be structured.

4.1.4 The installation tools

The installation tools place the burden of installation on the system rather than on the package being installed.

These tools are introduced to you here so that you can understand the environment into which your package will be placed. Manual pages for these tools are provided in Chapter "Reference pages", so that you can use them to test your package installation. The installation tools are:

- *pkgadd* installs a package.
- *pkgrm* removes a package.
- *pkgask* creates a file that contains an installer's response to prompts in the request script. This file is named on the *pkgadd* command line when a package is installed in noninteractive mode (simulated installation). It replaces the output of the request script.
- *pkgchk* checks the content and attribute information for packages to ensure that they were not corrupted during installation or on the medium.
- *pkginfo* and *pkgparam* display information about packages.

The system administrator can set parameters that control various aspects of installation in an administration file called the *admin* file. Refer to the manual entries in Chapter "Reference pages", for more information on these commands and on the *admin* file.

4.1.4.1 The package information files

Each of the six package information files will be described in the following pages. All of these files can be created using any editor. File formats are described in the following text and in full detail on the respective manual entry in Chapter "Reference pages". The six package information files are:

1. the *pkginfo* file
2. the *prototype* file
3. the *compver* file
4. the *copyright* file
5. the *depend* file
6. the *space* file

This section also describes the system-generated *pkgmap* file, which *pkgmk* creates and places on the installation medium. It is similar to the *prototype* file.

4.1.4.2 The pkginfo file

This required package information file describes characteristics of the package, such as the package abbreviation, full package name, package version, and package architecture. The definitions in this file can set values for all of the installation parameters defined in the *pkginfo* manual entry (see pkginfo - package characteristics file) found in Chapter "Reference pages".

The package installation procedure requires five mandatory parameters (see the following figure), these being:

PKG
: This parameter is assigned the package abbreviation, which may be no more than nine characters long.

NAME
: Specifies the full package name. The assigned value may be no more than 256 characters long.

ARCH
: Indicates the system architecture being used. The assigned value may be no more than 16 characters long. If a package supports multiple architectures, the identifiers (each up to 16 characters long) must be arranged alphabetically and separated by commas.

VERSION
: Version string. The assigned valued may be no more than 256 characters long.

CATEGORY
: Package affiliation. The assigned valued may be no more than 256 characters long.

Each entry in the file uses the following format to establish the value of a parameter:

PARAM="value"

If there are to be a number of diffferent installations on the same system, one of the para-meters *VERSION*, *PKG* or *ARCH* must be altered in the *pkginfo* file. This figure shows a sample *pkginfo* file.

```
PKG="pkgA"
NAME="My□Package□A"
ARCH="i386"
RELEASE="4.0"
VERSION="2"
VENDOR="MYCOMPANY"
CATEGORY="application"
ISTATES="S□2"
RSTATES="S□2"
```

The *pkginfo* and *pkgparam* commands can be used to access information in a *pkginfo* file.

Before defining the PKG, ARCH, and VERSION parameters, you need to know how *pkgadd* defines a package instance and the rules associated with naming a package. Refer to Section "Defining a package instance", before assigning values to these parameters.

4.1.4.3 The prototype file

This required package information file contains a list of the package contents. The *pkgmk* command uses the *prototype* file to identify the contents of a package and their location on the development machine when building the package.

You can create this file in two ways. As with all the package information files, you can use an editor to create a file named *prototype*. It should contain entries following the description given later in this chapter. You can also use the *pkgproto* command (see pkgproto - generate a prototype file) to automatically generate the file. To make use of the second method, you must have a copy of your package on your development machine that is structured exactly as you want it structured on the installation machine and all modes and permissions must be correct. If you are not going to use *pkgproto*, you do not need a structured copy of your package.

There are two types of entries in the *prototype* file: description lines and command lines.

The description lines

You must create one description line for each deliverable object that consists of several fields describing the object. This entry describes such information as mode, owner, and group for the object. You can also use this entry to accomplish the tasks listed below.

- You can override *pkgmk*'s placement of an object on a multiple-part package. (Refer to Section "Distributing packages over multiple volumes", for more details.)
- You can place objects into classes. (Refer to Section "Placing objects into classes", for details.)
- You can tell *pkgmk* where to find an object in your development directory structure and map that name to the correct placement on the installation machine. (Refer to the section entitled Mapping development pathnames to installation pathnames for details.)
- You can define an object as relocatable. (Refer to Section "Making package objects relocatable", for details.)
- You can define links. (Refer to Section "Creating the prototype file", for details.)

The generic format of the descriptive line is:

[part] ftype class pathname [major minor] [mode owner group]

Definitions for each field are as follows:

part

Designates the part in which an object should be placed. A package can be divided into a number of parts. This is typically useful of the package is to be split over a number of volumes (e.g. multiple floppy disks). A part is a collection of files and is the atomic unit by which a package is processed. A developer can choose the criteria for grouping files into a part (for example, by class). If not defined,

pkgmk decides in which part the object will be placed.

ftype
: Designates the file type of an object. Example file types are *f* (a standard executable or data file), *d* (a directory), *l* (a linked file), and *i* (a package information file). (Refer to the *prototype* manual entry in Chapter "Reference pages", for a complete list of file types.)

class
: Defines the class to which an object belongs. All objects must belong to a class. If the object belongs to no special class, this field should be defined as *none*.

pathname
: Defines the pathname which an object should have on the installation machine. If you do not begin this name with a slash, the object is considered to be relocatable. You can use the form *path1=path2* to map the location of an object on your development machine to the pathname it should have on the installation machine.

major/minor
: Defines the major and minor numbers for a block or character special device.

mode/owner/group
: Defines the mode, owner, and group for an object. If not defined, the defaults defined with the *default* command are assigned. If not defined and there are no defaults, the values *644 root other* are used.

This figure shows an example of this file with only description lines.

```
i□pkginfo
i□request
d□bin□ncmpbin□0755□root□other
f□bin□ncmpbin/dired=/usr/ncmp/bin/dired□0755□root□other
f□bin□ncmpbin/less=/usr/ncmp/bin/less□0755□root□other
f□bin□ncmpbin/ttype=/usr/ncmp/bin/ttype□0755□root□other
```

The line defining defaults for mode, owner and group must come before the file and directory definitions in order to be be applicable to the entire definition.

File and directory definitions must not include "//". Variables are recognized only if preceded and followed by a blank, a tab or a slash character "/". The first letter of a variable must be uppercase.

The command lines

There are four types of commands that can be embedded in the *prototype* file. They are:

1. search pathnames

   Specifies a list of directories (separated by white space) in which *pkgmk* should search when looking for package objects. pathnames is prepended to the basename of each object in the *prototype* file until the object is located.

2. include filename

   Specifies the pathname of another *prototype* file that should be merged into this one during processing. (Note that *search* requests do not span *include* files. Each *prototype* file should have its own *search* command defined, if one is needed.)

3. default mode owner group

   Defines the default *mode owner group* that should be used if this information is not supplied in a *prototype* entry that requires the information. (The defaults do not apply to entries in any *include* files. Each *prototype* should have its own *default* command defined, if one is needed.)

4. param=value

   Places the indicated parameter in the packaging environment. This allows you to expand a variable pathname so that *pkgmk* can locate the object without changing the actual object pathname. (This assignment will not be available in the installation environment.)

A command line must always begin with an exclamation point (*!*). Commands may have variable substitutions

embedded within them. This figure shows a sample *prototype* file with both description and command lines.

```
!PROJDIR=/usr/myname
!search□/usr/myname/bin□/usr/myname/src□/usr/myname/hdrs
!include□$PROJDIR/src/prototype
i□pkginfo
i□request
d□bin□ncmpbin□0755□root□other
f□bin□ncmpbin/dired=/usr/ncmp/bin/dired□0755□root□other
f□bin□ncmpbin/less=/usr/ncmp/bin/less□0755□root□other
f□bin□ncmpbin/ttype=/usr/ncmp/bin/ttype□0755□root□other
!default□755□root□bin
```

4.1.4.4 The compver file

This package information file defines previous (or future) versions of the package that are compatible with this version. Each line in the file consists of a string defining a version of the package with which the current version is compatible. The string must match the definition of the *VERSION* parameter in the *pkginfo* file of the package considered to be compatible. This figure shows an example of this file (for a vpkg package, see Section "The depend file").

```
1.3
1.0
```

4.1.4.5 The copyright file

This package information file contains the text of a copyright message that will be printed on the terminal at the time of package installation. The display is exactly as shown in the file. This figure shows an example of this file.

```
Copyright□(c)□Siemens□Nixdorf□Informationssysteme□1993
All□Rights□Reserved.
THIS□PACKAGE□CONTAINS□UNPUBLISHED□PROPRIETARY□SOURCE□CODE□OF□SNI.
The□copyright□notice□above□does□not□evidence□any
actual□or□intended□publication□of□such□source□code.
```

4.1.4.6 The depend file

This package information file defines software dependencies associated with the package. You can define three types of package dependencies with this file:

1. a *prerequisite package* (meaning this package depends on the existence of another package)
2. a *reverse dependency* (meaning another package depends on the existence of this package)
3. an *incompatible package* (meaning your package is incompatible with this one)

The generic format of a line in this file is:

```
type pkg name
□□□□□□□□□(arch)version
□□□□□□□□□(arch)version
```

Definitions for each field are as follows:

type
: Defines the dependency type. *P* indicates the named package is a prerequisite for installation. *I* indicates the named package is incompatible. *R* indicates a reverse dependency, that is, the named package requires that this package be on the system.

pkg
: Indicates the package abbreviation for the package.

name
: Specifies the full package name (used for display purposes only).

(arch)version
: Defines a particular instance of a package by defining the architecture and version. If *(arch)version* is not supplied, it means the entry refers to any instance of the package.

In a product dependency definition, the version specification must be exact; in other words, the arithmetic operators ">" and "<" and wildcards are not supported. If the *version* field is preceded by a tilde, the corresponding *compver* files are checked for compatible versions; thus in conjunction with the sample *compver* file in the following figure a *depend* file could include the specification: i386 ~1.3.

This figure shows an example of this file.

```
P□acu□□□Advanced□C□Utilities
□□□□□□□□□Issue□4□Version□1
P□cc□□□□C□Programming□Language
□□□□□□□□□(i386)□Issue□4□Version□1
R□vpkg□□Another□Vendor□Package
□□□□□□□□□(i386)□~1.3
```

If there is a tilde before the version string in the *depend* file of a dependent package, the *compver* file is examined. That means that in the event of a version change it is necessary to update the *compver* file in a package whose version is changing but is still compatible with one or more earlier versions.

4.1.4.7 The space file

This package information file defines disk space requirements for the target environment beyond that which is used by objects defined in the *prototype* file—for example, files that will be dynamically created at installation time. It should define the maximum amount of additional space that a package will require.

The generic format of a line in this file is:

pathname□blocks□inodes

Definitions for each field are as follows:

name
: Names a directory in which there are objects that will require additional space. The name may be the mount point for a filesystem. Names that do not begin with a slash (/) indicate relocatable directories.

blocks
: Defines the number of 512 byte disk blocks required for installation of the files and directory entries contained in the pathname. (Do not include file system dependent disk usage.)

inodes
: Defines the number of inodes required for installation of the files and directory entries contained in *name*.

This figure shows an example of this file.

```
#□extra□space□required□by□config□data□which□is
#□dynamically□loaded□onto□the□system
data□□500□□1
```

4.1.4.8 The pkgmap file

The *pkgmk* command creates the *pkgmap* file when it processes the *prototype* file. This new file contains all of the information in the *prototype* file plus three new fields for each entry. These fields are *size* (file size in bytes), *cksum* (checksum of file), and *modtime* (last time of modification). All command lines defined in the *prototype* file are executed as *pkgmk* creates the *pkgmap* file. The *pkgmap* file is placed on the installation medium. The *prototype* file is not. Refer to the *pkgmap* manual entry (refer to pkgmap - package contents description file) in Chapter "Reference pages", for more details about this file.

4.1.5 The installation scripts

The *pkgadd* command automatically performs all of the actions necessary to install a package, using the package information files as input. As a result, you do not have to supply any packaging scripts. However, if you want to customize the installation procedures for your package needs, the following three types of scripts can be used:

1. request script

   Solicits administrator interaction during package installation for the purpose of assigning or redefining

environment parameter assignments. Interactions of this type must always be defined in request scripts.

2. class action scripts

   Define an action or set of actions that should be applied to a class of files during installation or removal. You define your own classes or you can use one of three standard classes (*sed*, *awk*, *build* and *CONFIG.prsv*). See Section "Placing objects into classes", for details on how to define a class.

3. procedure scripts

   Specifies a procedure to be invoked before or after the installation or removal of a package. The four procedure scripts are *preinstall*, *postinstall*, *preremove*, and *postremove*.

You decide which type of script to use based on when you want the script to execute. To help you with this assessment, script processing is discussed next, followed by a description of parameters available to packaging scripts, how to get information about a package for your scripts, and script exit codes. After that, each type of script is described in detail.

All installation scripts must be executable by the *sh* shell (for example, a shell script or a program executable).

4.1.5.1 Script processing

You can customize the actions taken during installation by delivering installation scripts with your package. The decision on which type of script to use to meet a need depends upon when the action of the script is needed during the installation process. As a package is installed, *pkgadd* performs the following steps:

- Executes the request script.

  This is the only point at which your package can solicit input from the installer.

- Executes the preinstall script.
- Installs the package objects.

  Installation occurs class-by-class and class action scripts are executed accordingly. The list of classes operated upon and the order in which they should be installed is initially defined with the *CLASSES* parameter in your *pkginfo* file. However, your request script can change the value of *CLASSES*.

- Executes the *postinstall* script.

  When a package is being removed, *pkgrm* performs these steps:

- Executes the *preremove* script.
- Executes the removal class action scripts.

  Removal also occurs class-by-class. As with the installation class action scripts, if more than one removal script exists, they are processed in the reverse order in which the classes were listed in the *CLASSES* parameter at the time of installation.

- Executes the *postremove* script.

The request script is not processed at the time of package removal. However, its output (a list of parameter values) is saved and so is available to removal scripts.

4.1.5.2 Installation parameters

These following four groups of parameters are available to all installation scripts. Some of the parameters can be modified by a request script, others cannot be modified at all.

- The four system parameters that are part of the installation software (see below for a description of these). None of these parameters can be modified by a package.
- The 20 standard installation parameters defined in the *pkginfo* file. Of these, a package can only modify the *CLASSES* parameter. (The standard installation parameters are described in detail in the *pkginfo* manual entry in the "Reference pages" chapter.)
- You can define your own installation parameters by assigning a value to them in the *pkginfo* file. Such a parameter must be alphanumeric with an initial capital letter. Any of these parameters can be changed by a request script.

- Your request script can define new parameters by assigning values to them and placing them into the installation environment, as shown in the figure Placing parameters into the installation environment.

The four installation parameters that can be accessed by installation scripts are described below:

PATH
: Specifies the search list used by *sh* to find commands; on script invocation, *PATH* is set to */sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin*.

UPDATE
: Indicates that the current installation is intended to update the system. Automatically set to *yes* if the package being installed is overwriting a version of itself.

PKGINST
: Specifies the instance identifier of the package being installed. If another instance of the package is not already installed, the value will be the package abbreviation. Otherwise, it is the package abbreviation followed by a suffix, such as *pkg.1*.

  (Multiple variations of the same package can reside simultaneously on the installation medium, as well as on the installation machine. Each variation is known as a package instance and assigned an instance identifier. See "Defining a package instance" for more details.)

PKGSAV
: Specifies the directory where files can be saved for use by removal scriptsor where previously saved files may be found.

The */var/sadm/install/admin/default* file transfers installation defaults to the system. If no user is specified for the *mail=* parameter, user *root* will be notified.

4.1.5.3 Getting package information for a script

There are two commands that can be used from your scripts to solicit information about a package.

1. The *pkginfo* command returns information about software packages, such as the instance identifier and package name.
2. The *pkgparam* command returns values only for the parameters requested.

The *pkginfo* and *pkgparam* (*(1)* and *(4)*) manual entries in the "Reference pages" chapter give details for these tools.

4.1.5.4 Exit codes for scripts

Each script must exit with one of the following exit codes:

0
: Successful completion of script.

1
: Fatal error. Installation process is terminated at this point.

2
: Warning or possible error condition. Installation will continue. A warning message will be displayed at the time of completion.

3
: Script was interrupted and possibly left unfinished. Installation terminates at this point.

10
: System should be rebooted when installation of all selected packages is completed. (This value should be added to one of the single-digit exit codes described above.)

20
: The system should be rebooted immediately upon completing installation of the current package. (This value should be added to one of the single-digit exit codes described above.)

4.1.5.5 The request script

The request script solicits interaction during installation and is the only place where your package can interact

directly with the installer. It can be used, for example, to ask the installer if optional pieces of a package should be installed.

The output of a request script must be a list of parameters and their values. This list can include any of the parameters you created in the *pkginfo* file and the *CLASSES* parameter. The list can also introduce parameters that have not been defined elsewhere.

When your request script assigns values to a parameter, it must then make those values available to the installation environment for use by *pkgadd* and also by other packaging scripts. The following example shows a request script segment that performs this task for the four parameters *CLASSES*, *NCMPBIN*, *EMACS*, and *NCMPMAN*.

Placing parameters into the installation environment

```
#□make□parameters□available□to□installation□service
#□and□any□other□packaging□script□we□might□have
cat□>$1□<<!
CLASSES='$CLASSES
NCMPBIN='$NCMPBIN
EMACS='$EMACS
NCMPMAN='$NCMPMAN
!
```

Request script naming conventions

There can only be one request script per package and it must be named *request*.

Request script usage rules

- The request script can not modify any files. It is intended only to interact with users and to create a list of parameter assignments based upon that interaction. (To enforce this restriction, the request script is executed as the nonprivileged user *install*.)
- *pkgadd* calls the request script with one argument that names the file to which the output of this script will be written.
- The parameter assignments should be added to the installation environment for use by *pkgadd* and other packaging scripts (as shown in the figure Placing parameters into the installation environment).
- System parameters and standard installation parameters, except for the *CLASSES* parameter, cannot be modified by a request script. Any of the other parameters available can be changed.
- The format of the output list should be *PARAMETER*="*value*". For example:

  CLASSES="none class1"
- The list should be written to the file named as the argument to the request script.
- The user's terminal is defined as standard input to the request script.
- The request script is not executed during package removal. However, the parameter values assigned in the script are saved and are available during removal.

4.1.5.6 The class action script

The class action script defines a set of actions to be executed during installation or removal of a package. The actions are performed on a group of pathnames based on their class definition.

Class action script naming conventions

The name of a class action script is based on which class it should operate and whether those actions should occur during package installation or removal. The two name formats are:

i.class
: (operates on pathnames in the indicated class during package installation)

r.class
: (operates on pathnames in the indicated class during package removal)

For example, the name of the installation script for a class named *class1* would be *i.class1* and the removal script would be named *r.class1*.

Class action script usage rules

- Class action scripts are executed as *uid=root* and *gid=other*.
- If a package spans more than one volume, the class action script will be executed once for each volume that contains at least one file belonging to the class. Consequently, each script must be 'multiply executable'. This means that executing a script any number of times with the same input must produce the same results as executing the script only once.

  The installation service relies upon this condition being met.
- The script is not executed if no files in the given class exist on the current volume.
- *pkgadd* (and *pkgrm*) creates a list of all objects listed in the *pkgmap* file that belong to the class. As a result, a class action script can only act upon pathnames defined in the *pkgmap* and belonging to a particular class.
- A class action script should never add, remove, or modify a pathname or system attribute that does not appear in the list generated by *pkgadd* unless by use of the *installf* or *removef* command.
- When the class action script executes for the last time (meaning the input pathname is the last path on the last volume containing a file of this class), it is executed with the keyword argument *ENDOFCLASS*. This flag allows you to include post-processing actions into your script.

Installation of classes

The following steps outline the system actions that occur when a class is installed. The actions are repeated once for each volume of a package as that volume is being installed.

1. *pkgadd* creates a pathname list.

   pkgadd creates a list of pathnames upon which the action script will operate. Each line of this list consists of source and destination pathnames, separated by white space. The source pathname indicates where the object to be installed resides on the installation volume and the destination pathname indicates the location on the installation machine where the object should be installed. The contents of the list is restricted by the following criteria:
   - The list contains only pathnames belonging to the associated class.
   - Directories, named pipes, character/block devices, and symbolic links are included in the list with the source pathname set to */dev/null*. They are automatically created by *pkgadd* (if not already in existence) and given proper attributes (mode, owner, group) as defined in the *pkgmap* file.
   - Linked files are not included in the list, that is, files where *ftype* is *l*. (*ftype* defines the file type and is defined in the *prototype* file. Links in the given class are created in Step 4.)
   - If a pathname already exists on the target machine and its contents are no different from the one being installed, the pathname will not be included in the list.
   - To determine this, *pkgadd* compares the *cksum*, *modtime*, and *size* fields in the installation software database with the values for those fields in your *pkgmap* file. If they are the same, it then checks the actually file on the installation machine to be certain it really has those values. If the field values are the same and are correct, the pathname for this object will not be included in the list.
2. If there is no class action script, the pathnames are copied to the target machine.

   If no class action script is provided for installation of a particular class, the pathnames in the generated list will simply be copied from the volume to the appropriate target location.
3. If there is a class action script, the script is executed.

   The class action script is invoked with standard input containing the list generated in Step 1. If this is the last volume of the package and there are no more objects in this class, the script is executed with the single argument of *ENDOFCLASS*.
4. *pkgadd* performs a content and attribute audit and creates links.

After successfully executing Step 2 or 3, an audit of both content and attribute information is performed on the list of pathnames. *pkgadd* creates the links associated with the class automatically. Detected attribute inconsistencies are corrected for all pathnames in the generated list.

Removal of classes

Objects are removed class-by-class. Classes that exist for a package, but are not listed in the *CLASSES* parameter are removed first (for example, an object installed with the *installf* command). Classes that are listed in the *CLASSES* parameter are removed in reverse order. The following steps outline the system actions that occur when a class is removed:

1. *pkgrm* creates a pathname list.

   pkgrm creates a list of installed pathnames that belong to the indicated class. Pathnames referenced by another package are excluded from the list unless their *ftype* is *e* (meaning the file should be edited upon installation or removal).

   If a pathname is referenced by another package, it will not be removed from the system. However, it may be modified to remove information placed in it by the package being removed.

2. If there is no class action script, the pathnames are removed.

   If your package has no removal class action script for the class, all of the pathnames in the list generated by *pkgrm* will be removed.

   You should always assign a class for files with an *ftype* of *e* (editable) and have an associated class action script for that class. Otherwise, they will be removed at this point, even if the pathname is shared with other packages.

3. If there is a class action script, the script is executed.

   pkgrm invokes the class action script with standard input containing the list generated in Step 1.

4. *pkgrm* performs an audit.

   Upon successful execution of the class action script, knowledge of the pathnames is removed from the system unless a pathname is referenced by another package.

4.1.5.7 The special system classes

The system provides four special classes. They are:

1. The *sed* class (provides a method for using *sed* instructions to edit files upon installation and removal).
2. The *awk* class (provides a method for using *awk* instructions to edit files upon installation and removal).
3. The *build* class (provides a method to dynamically construct a file during installation).
4. The *CONFIG.prsv* class (installs a file only if it is not yet present).

The sed class script

The *sed* installation class provides a method of installing and removing objects that require modification to an existing object on the target machine. A *sed* class action script delivers *sed* instructions in the format shown in the following figure. You can give instructions that will be executed during either installation or removal. Two commands indicate when instructions should be executed. *sed* instructions that follow the *!install* command are executed during package installation, and those that follow the *!remove* command are executed during package removal. It does not matter in which order the commands are used in the file.

The *sed* class action script executes automatically at installation time if a file belonging to class *sed* exists. The name of the *sed* class file should be the same as the name of the file upon which the instructions will be executed.

```
#□comment,□which□may□appear□on□any□line□in□the□file
!install
#□sed(1)□instructions□which□are□to□be□invoked□during
#□installation□of□the□object
[address□[,addres]]□function□[arguments]
□□□.□.□.
```

```
!remove
#□sed(1)□instructions□to□be□invoked□during□the□removal□process
[address□[,address]]□function□[arguments]
□□□.□.□.
```

address, *function*, and *arguments* are as defined in the manual entry *sed(1)* in [6].

The awk class script

The *awk* installation class provides a method of installing and removing objects that require modification to an existing object on the target machine. Modifications are delivered as *awk* instructions in an *awk* class action script.

The *awk* class action script executes automatically at the time of installation if a file belonging to class *awk* exists. Such a file contains instructions for the *awk* class script in the format shown in the following figure. Two commands indicate when instructions should be executed. *awk* instructions that follow the *!install* command are executed during package installation, and those that follow the *!remove* command are executed during package removal. It does not matter in which order the commands are used in the file.

The name of the *awk* class file should be the same as the name of the file upon which the instructions will be executed.

```
#□comment,□which□may□appear□on□any□line□in□the□file
!install
#□awk(1)□program□to□install□changes
□□□□.□.□.□(awk□program)
!remove
#□awk1(1)□program□to□remove□changes
□□□□.□.□.□(awk□program)
```

The file to be modified is used as input to *awk* and the output of the script ultimately replaces the original object. Parameters may not be passed to *awk* using this syntax.

The build Class Script

The *build* class installs or removes objects by executing instructions that create or modify the object file. These instructions are delivered as a *build* class action script.

The name of the instruction file should conform to standard Reliant UNIX system naming conventions.

The *build* class action script executes automatically at installation time if a file belonging to class *build* exists.

A *build* script must be executable by *sh*. The script's output becomes the new version of the file as it is built.

See Section "Case 5c: Modifying existing data files - using the build class", for an example *build* class action script.

The CONFIG.prsv class script

The *CONFIG.prsv* class preserves existing configuration files.

The associated script first checks whether the file that is to be installed already exists. If it does, the existing ffile will not be overwritten. If it does not, it will be installed in the usual way. This mechanism cannot be used to merge old and new files; it may be possible to used an *awk* or *sed* class action script for this purpose.

4.1.5.8 The procedure script

The procedure script gives a set of instructions that are performed at particular points in installation or removal. Four possible procedure scripts are described below.

Naming conventions for procedure scripts

The four procedure scripts must use one of the names listed below, depending on when these instructions are to be executed.

preinstall
(executes before class installation begins)

postinstall
(executes after all volumes have been installed)

preremove
(executes before class removal begins)

postremove
(executes after all classes have been removed)

Procedure script usage rules

- Procedure scripts are executed as *uid=root* and *gid=other*.
- Each script should be multiply executable since it will be executed once for each volume in a package. This means that executing a script any number of times with the same input will produce the same results as executing the script only once.
- Each installation procedure script that will add or modify a pathname must use the *installf* or *removef* command to notify *pkgadd* that it will do so. After all additions or modifications are complete, this command should be invoked with the *–f* option to indicate all additions and modifications are complete. (See the manual entries installf - add a file to the software installation database or removef remove a file from software database in Chapter "Reference pages", for details and examples.)
- Each removal procedure script that will add or modify a pathname must use the *installf* or *removef* command to notify *pkgrm* that it will do so. After removal is complete, this command should be invoked with the *-f* option to indicate all removals have been completed. (See the manual entries installf - add a file to the software installation database or removef remove a file from software database in Chapter "Reference pages", for details and examples.)

The *installf* and *removef* commands must be used for all path name which are not auto-matically associated with any pathnames listed in the *pkgmap* file.

installf- and *removef* are not to be uniquely associated with installation and removal respectively.

4.1.6 Basic steps of packaging

Since what steps you take to create a package depends on how customized your package will be, it is difficult to give you a step-by-step guide on how to proceed. Your first step should be to plan your packaging. For example you must decide on which package information files and scripts your package needs.

The following list presents an overview of some of the steps you might use in a packaging scenario. Not all of these steps are required and there exists no mandated order for their execution (although you must have all of your package objects together before executing *pkgmk*). The remainder of this chapter gives procedural information for each step.

This list, and the following procedures, are intended as a guideline and should not replace reading the rest of this chapter to learn what options are available to your package or replace your own individualized planning.

- Assign a package abbreviation.

  Every package installed in a Version 5.41 environment must have a package abbreviation.
- Define a package instance.

  You must decide on values for the three package parameters that will make each package instance unique. (You need to understand what a package instance is, how it is defined, what the instance identifier is, and how to use that identifier. All of this is covered in the procedure "Defining a package instance".)
- Place your objects into classes.

  You must decide on what installation classes you are going to use before you can create the *prototype* file and also before you can write your class action scripts.
- Set up a package and its objects as relocatable.

  Package objects can be delivered with either fixed locations, meaning that their location is defined by the package and cannot be changed, or with relocatable locations, meaning that they have no absolute location requirements. All of a package or parts of a package can be defined as relocatable. You should

decide if package objects will have fixed locations or be relocatable before you write any installation scripts and before you create the *prototype* file.

- Decide which installation scripts your package needs.

  You must assess the needs of your package beyond the actions provided by *pkgadd* and decide on which type of installation scripts will allow you to deliver your customized actions.
- Define package dependencies.

  You must decide if your package has dependencies on other packages and if any other packages depend on yours.
- Write a copyright message.

  You must decide if your package requires a copyright message to appear as it is being installed (and removed) and, if so, you must write that message.
- Create the *pkginfo* file.

  You must create a *pkginfo* file before executing *pkgmk*. It defines basic information concerning the package and can be created with any editor as long as it follows the format described earlier in this chapter and in the *pkginfo* manual entry in the "Reference pages" chapter.
- Create the *prototype* file.

  This file is required and must be created before you execute *pkgmk*. It lists all of the objects that belong to a package and information about each object (such as its file type and to which class it belongs). You can create it using any editor and you must follow the format described earlier in this chapter and in the *prototype* manual entry in the "Reference pages" chapter. You can also use the *pkgproto* command to generate a *prototype* file.
- Distribute packages over multiple volumes.

  pkgmk automatically distributes packages over multiple volumes. You must decide if you want to leave those calculations up to *pkgmk* or customize package placement on multiple volumes.
- Create the package.

  Create the package using the *pkgmk* command, which copies objects from the development machine to the installation medium, puts them into the proper structure, and automatically spans them across multiple volumes, if necessary.

  This is always the last step of packaging, unless you want to create a datastream structure for your package. If so, you must execute *pkgtrans* after creating a package with *pkgmk*.

4.1.7 Assigning a package abbreviation

Each package installed on a Version 5.41 machine must have a package abbreviation assigned to it. This abbreviation is defined with the *PKG* parameter in the *pkginfo* file.

A valid package abbreviation must meet the criteria defined below:

- It must start with an alphabetic character.
- Additional characters may be alphanumeric and contain the two special characters + and -.
- It cannot be longer than nine characters.
- Reserved names are *install*, *new*, and *all*.

4.1.8 Defining a package instance

The same software package can differ by version or architecture or both. Multiple variations of the same package can reside simultaneously on the same machine. Files are jointly assigned to the different variations. Each variation is known as a package instance. *pkgadd* assigns a package identifier to each package instance at the time of installation. The package identifier is the package abbreviation with a numerical suffix. This identifier distinguishes an instance from any other package, including other instances of the same package.

4.1.8.1 Identifying a package instance

Three parameters defined in the *pkginfo* file combine to uniquely identify each instance. You cannot assign identical values for all three parameters for two instances of the same package installed in the same target environment. These parameters are:

PKG
: defines the software package abbreviation and remains constant for every instance of a package.

VERSION
: defines the software package version.

ARCH
: defines the software package architecture.

For example, you might identify two identical versions of a package that run on different hardware as:

Instance□#1□□□□□□□□□□□□□Instance□#2
PKG="abbr"□□□□□□□□□□□□□□PKG="abbr"
VERSION="release□1"□□□□VERSION="release□1"
ARCH="i386"□□□□□□□□□□□□□ARCH="R3000"

Two different versions of a package that run on the same hardware might be identified as:

Instance□#1□□□□□□□□□□□□□Instance□#2
PKG="abbr"□□□□□□□□□□□□□□PKG="abbr"
VERSION="release□1"□□□□VERSION="release□2"
ARCH="i386"□□□□□□□□□□□□□ARCH="i386"

The instance identifier, assigned by *pkgadd*, maps the three pieces of information that identify an instance to one name consisting of the package abbreviation plus a suffix. The first instance of a package installed on a system does not have a suffix and so its instance identifier will be the package abbreviation. Subsequent instances receive a suffix, beginning with *.2*. An instance is given the lowest integer extension available and so may not correspond to the order in which a package was installed. For example, if *mypkg.2* was deleted after *mypkg.3* was installed, the next instance to be added will be named *mypkg.2*. Because the number of instances of a particular package can vary from machine to machine, the instance identifier can also vary.

pkgmk also assigns an instance identifier to a package as it places it on the installation medium if one or more instances of a package already exists. That identifier bears no relationship to the identifier assigned to the same package on the installation machine.

4.1.8.2 Accessing the instance identifier in your scripts

Because the instance identifier is assigned at the time of installation and will differ from machine to machine, you should use the *PKGINST* system parameter to reference your package in your installation scripts.

4.1.9 Writing your installation scripts

You should read Section "The installation scripts", to learn what types of scripts you can write and how to write them.

Remember, you are not required to write any installation scripts for a Version 5.41 package. The *pkgadd* command performs all of the actions necessary to install your package, using the information you supply with the package information files. Any installation script that you write will be used to perform customized actions beyond those executed by *pkgadd*.

Be certain that every installation script being delivered with your package has an entry in the *prototype* file. The file type should be *i*.

4.1.10 Making package objects relocatable

Package objects can be delivered either with fixed locations, meaning that their location on the installation machine is defined by the package and cannot be changed, or as relocatable, meaning that they have no absolute location requirements on the installation machine. The location for relocatable package objects is determined during the installation process.

You can define two types of relocatable objects: collectively relocatable and individually relocatable. All collectively relocatable objects are placed relative to the same directory once the relocatable root directory is

established. Individually relocatable objects are not restricted to the same directory location as collectively relocatable objects.

4.1.10.1 Defining collectively relocatable objects

Follow these steps to define package objects as collectively relocatable:

1. Define a value for the *BASEDIR* parameter.

   Put a definition for the *BASEDIR* parameter in your *pkginfo* file. This parameter names a directory where relocatable objects will be placed by default. If you supply no value for *BASEDIR*, no package objects will be considered as collectively relocatable.

2. Define objects as collectively relocatable in the *prototype* file.

   An object is defined as collectively relocatable by using a relative pathname in its entry in the *prototype* file. A relative pathname does not begin with a slash. For example, *src/myfile* is a relative pathname, while */src/myfile* is a fixed pathname.

   A package can deliver some objects with relocatable locations and others with fixed locations.

   All objects defined as collectively relocatable will be put under the same root directory on the installation machine. The root directory value will be one of the following (and in this order):

   - the installer's response to *pkgadd* when asked where relocatable objects should be installed
   - the value of *BASEDIR* as it is defined in the installer's */var/sadm/install/admin/** file(s) for a given installation (the *BASEDIR* value assigned to the file with *pkgadd -a* overrides the value in the *pkginfo* file)
   - the value of *BASEDIR* as it is defined in your *pkginfo* file (this value is used only as a default in case the other two possibilities have not supplied a value)

4.1.10.2 Defining individually relocatable objects

A package object is defined as individually relocatable by using a variable in its pathname definition in the *prototype* file. Your request script must query the installer on where such an object should be placed and assign the response value to the variable. *pkgadd* will expand the pathname based on the output of your request script at the time of installation. Case 1: Installing objects conditionally in Section "Package installation case studies", shows an example of the use of variable pathnames and the request script needed to solicit a value for the base directory.

4.1.11 Placing objects into classes

Installation classes allow a series of actions to be performed on a group of package objects at the time of their installation or removal. You place objects into a class in the *prototype* file. All package objects must be given a class, although the class of *none* may be used for objects that require no special action.

The installation parameter *CLASSES*, defined in the *pkginfo* file, is a list of classes to be installed (including the *none* class). Objects defined in the *prototype* file that belong to a class not listed in this parameter will not be installed. The actions to be performed on a class (other than simply copying the components to the installation machine) are defined in a class action script. These scripts are named after the class itself.

For example, to define and install a group of objects belonging to a class named *class1*, follow these steps:

- Define the objects belonging to *class1* as such in their *prototype* file entry. For example,

  f□class1□/usr/src/myfile
  f□class1□/usr/src/myfile2

- Ensure that the *CLASSES* parameter in the *pkginfo* file has an entry for *class1*. For example,

  CLASSES="class1□class2□none"

- Ensure that a class action script exists for this class. An installation script for a class named *class1* would be named *i.class1* and a removal script would be named *r.class1*.

  If you define a class but do not deliver a class action script, the only action taken for that class will be to copy components from the installation medium to the installation machine.

In addition to the classes that you can define, the system provides four standard classes for your use. The *sed*

class provides a method for using *sed* instructions to edit files upon package installation and removal. The *awk* class provides a method for using *awk* instructions to edit files upon package installation and removal. The *build* class provides a method to dynamically construct a file during package installation. The *CONFIG.prsv* class provides a method for leaving existing files as they are.

4.1.12 Defining package dependencies

Package dependencies and incompatibilities can be defined with two of the optional package information files. Delivering a *compver* file lets you name versions of your package that are compatible with the one being installed. Delivering a *depend* file lets you define three types of dependencies associated with your package. These dependency types are:

1. a prerequisite package,

   which means that your package depends on the existence of another package.
2. a reverse dependency,

   which means that another package depends on the existence of your package. This type has no effect in a Reliant UNIX 5.43 environment.
3. an incompatible package,

   which means that your package is incompatible with this one.

Refer to Section "The depend file", and to Section "The compver file", or study the manual entries compver - compatible versions file and depend - software package dependencies file in Chapter "Reference pages", for details on the formats of these files.

Be certain that your *depend* and *compver* files have entries in the *prototype* file. The file type should be *i* (for package information file).

4.1.13 Writing a copyright message

To deliver a copyright message, you must create a copyright file named *copyright*. The message will be displayed exactly as it appears in the file (no formatting) as the package is being installed and as it is being removed. Refer to Section "The copyright file", or the manual entry copyright copyright information file in Chapter "Reference pages", for more detail.

Be certain that your *copyright* file has an entry in the *prototype* file. Its file type should be *i* (for package information file).

4.1.14 Reserving additional space on the installation machine

pkgadd assures that there is enough disk space to install your package, based on the object definitions in the *pkgmap* file. However, sometimes your package will require additional disk space beyond that needed by the objects defined in the *pkgmap* file. For example, your package might create a file during installation. *pkgadd* checks for additional space when you deliver a *space* file with your package. Refer to Section "The space file", or to the manual entry space - disk space requirement file in Chapter "Reference pages", for details on the format of this file.

Be certain that your *space* file has an entry in the *prototype* file. Its file type should be *i* (for package information file).

4.1.15 Creating the pkginfo file

The *pkginfo* file establishes values for parameters that describe the package and is a required package component. The format for an entry in this file is:

PARAM="value"

PARAM can be any of the 19 standard parameters described in the *pkginfo* manual entry in the "Reference pages" chapter. You can also create your own package parameters simply by assigning a value to them in this file. Your parameter names must begin with a capital letter followed by either upper or lowercase letters.

The following five parameters are required:

PKG

(package abbreviation)

NAME
(full package name)

ARCH
(package architecture)

VERSION
(package version)

CATEGORY
(package category)

The *CLASSES* parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the *none* class must be defined in the *CLASSES* parameter before objects belonging to that class will be installed.

You can choose to define the value of *CLASSES* with a request script and not to deliver a value in the *pkginfo* file.

4.1.16 Creating the prototype file

The *prototype* file is a list of package contents and is a required package component.

You can create the *prototype* file by using any editor and following the format described in Section "The prototype file", and in the manual entry prototype - package information file in Chapter "Reference pages". You can also use the *pkgproto* command to create one automatically.

4.1.16.1 Creating the file manually

While creating the *prototype* file, you must at the very least supply the following three pieces of information about an object:

1. The object's type

   All of the possible object types are defined in the *prototype* manual entry in the "Reference pages" chapter. *f* (for a data file), *l* (for a linked file), and *d* (for a directory) are examples of object types.

2. The object's class

   All objects must be assigned a class. If no special handling is required, you can assign the class *none*.

3. The object's pathname

   The pathname can define a fixed pathname such as */mypkg/src/filename*, a collectively relocatable pathname such as *src/filename*, and an individually relocatable pathname such as *$BIN/filename* or */opt/$PKGINST/filename*.

Creating links

To define links you must do the following in the *prototype* entry for the linked object:

- Define its *ftype* as *l* (a link) or *s* (a symbolic link).
- Define its pathname with the format *path1=path2* where *path1* is the destination and *path2* is the source file.

Mapping development pathnames to installation pathnames

If your development area is in a different structure than you want the package to be in on the installation machine, you can use the *prototype* entry to map one pathname to the other. You use the *path1=path2* format for the pathname as is used to define links. However, if the *ftype* is not defined as *l* or *s*, *path1* is interpreted as the pathname you want the object to have on the installation machine and *path2* is interpreted as the pathname the object has on your development machine.

For example, your project might require a development structure that includes a project root directory and numerous *src* directories. However, on the installation machine you might want all files to go under a package root directory and for all *src* files to be in one directory. So, a file on your machine might be named */projdir/srcA/filename*. If you want that file to be named */pkgroot/src/filename* on the installation machine, your

prototype entry for this file might look like this:

f class1 /pkgroot/src/filename=/projdir/srcA/filename

Defining objects for pkgadd to create

You can use the *prototype* file to define objects that are not actually delivered on the installation medium. *pkgadd* creates objects with the following *ftype*s if they do not already exist at the time of installation:

d
: directories

x
: exclusive directories

l
: linked files

s
: symbolically linked files

p
: named pipes

c
: character special devices

b
: block special devices

To request that one of these objects be created on the installation machine, you should add an entry for it in the *prototype* file using the appropriate *ftype*.

For example, if you want a directory created on the installation machine, but do not want to deliver it on the installation medium, an entry for the directory in the *prototype* file is sufficient. An entry such as the one shown below will cause the directory to be created on the installation machine, even if it does not exist on the installation medium.

d none /directoryA 644 root other

Using the command lines

There are four types of command that you can put into your *prototype* file. They allow you to do the following:

1. Nest *prototype* files (the *include* command)
2. Define directories for *pkgmk* to look in when attempting to locate objects as it creates the package (the *search* command)
3. Set a default value for *mode owner group* (the *default* command). If all or most of your objects have the same values, using the *default* command will keep you from having to define these values for every entry in the *prototype* file.
4. Assign a temporary value for variable pathnames to tell *pkgmk* where to locate these relocatable objects on your machine (with *param*=*value*)

4.1.16.2Creating the file using pkgproto

The *pkgproto* command scans your directories and generates a *prototype* file. *pkgproto* cannot assign *ftype*s of *v* (volatile files), *e* (editable files), or *x* (exclusive directories). You can edit the *prototype* file and add these *ftype*s, as well as perform any other fine-tuning you require (for example, adding command lines or classes).

pkgproto writes its output to the standard output. To create a file, you should redirect the output to a file. The examples shown in this section do not perform redirection in order to show you what the contents of the file would like.

Creating a basic prototype

The standard format of *pkgproto* is

pkgproto□path□[□.□.□.□]

where *path* is the name of one or more paths to be included in the *prototype* file. If *path* is a directory, then entries are created for the contents of that directory as well.

With this form of the command, all objects are placed into the *none* class and are assigned the same *mode owner group* as exists on your machine. The following example shows *pkgproto* being executed to create a file for all objects in the directory */usr/bin*:

```
$ pkgproto /usr/bin
d none /usr/bin 755 bin bin
f none /usr/bin/datei1 755 bin bin
f none /usr/bin/datei2 755 bin bin
f none /usr/bin/datei3 755 bin bin
f none /usr/bin/datei4 755 bin bin
f none /usr/bin/datei5 755 bin bin
$
```

To create a *prototype* file that contains the output of the example above, you would execute

```
pkgproto /usr/bin > prototype
```

If no pathnames are supplied when executing *pkgproto*, standard input (*stdin*) is assumed to be a list of paths. Refer to the *pkgproto* manual entry in the "Reference pages" chapter for details on this usage.

Assigning objects to a class

You can use the *–c class* option of *pkgproto* to assign objects to a class other than *none*. When using this option, you can only name one class. To define multiple classes in a *prototype* file created by *pkgproto*, you must edit the file after its creation.

The following example is the same as above except the objects have been assigned to *class1*.

```
$ pkgproto -c class1 /usr/bin
d class1 /usr/bin 755 bin bin
f class1 /usr/bin/file1 755 bin bin
f class1 /usr/bin/file2 755 bin bin
f class1 /usr/bin/file3 755 bin bin
f class1 /usr/bin/file4 755 bin bin
f class1 /usr/bin/file5 755 bin bin
$
```

Renaming pathnames with pkgproto

You can use a *path1=path2* format on the *pkgproto* command line to give an object a different pathname in the *prototype* file than it has on your machine. You can, for example, use this format to define relocatable objects in a *prototype* file created by *pkgproto*.

The following example is like the others shown in this section, except that the objects are now defined as *bin* (instead of */usr/bin*) and are thus relocatable.

```
$ pkgproto -c class1 /usr/bin=bin
d class1 bin 755 bin bin
f class1 bin/datei1 755 bin bin
f class1 bin/datei2 755 bin bin
f class1 bin/datei3 755 bin bin
f class1 bin/datei4 755 bin bin
f class1 bin/datei5 755 bin bin
$
```

pkgproto and links

pkgproto detects linked files and creates entries for them in the *prototype* file. If multiple files are linked together, it considers the first path encountered the source of the link.

If you have symbolic links established on your machine but want to generate an entry for that file with an *ftype* of *f* (file), then use the *–i* option of *pkgproto*. This option creates a file entry for all symbolic links.

4.1.17 Distributing packages over multiple volumes

As packager, you no longer need to worry about placing package components on multiple volumes. *pkgmk* performs the calculations and actions necessary to organize a multiple volume package. As *pkgmk* creates your package, it will prompt you to insert a new volume as often as necessary to distribute the complete package over multiple volumes.

However, you can use the optional *part* field in the *prototype* file to define in which part you want an object to be placed. (See section The description lines.) A number in this field overrides *pkgmk* and forces the placement of the component into the part given in the field. Note again that there is a one-to-one correspondence between parts and volumes for removable media formatted as file systems.

4.1.18 Creating a package with pkgmk

pkgmk takes all of the objects on your machine (as defined in the *prototype* file), puts them in the fixed directory format and copies everything to the installation medium.

To package your software, execute

pkgmk [–d symbolic-name] [–f filename]

You must use the *–d* option to name the device onto which the package should be placed. *symbolic-name* can be a directory pathname or the identifier for a disk. The default device is the installation spool directory.

pkgmk looks for a file named *prototype*. You can use the *–f* option to specify a package contents file named something other than *prototype*. This file must be in the *prototype* format.

For example, executing *pkgmk –d symbolic-name* creates a package based on a file named *prototype* in your current working directory. The package will be formatted and copied to the diskette in the device with the symbolic name *symbolic-name*.

4.1.18.1 Creating a package instance

pkgmk will create a new instance of a package if one already exists on the device to which it is writing. It will assign the package an instance identifier. Use the *–o* option of *pkgmk* to overwrite an existing instance of a package rather than to create a new one.

4.1.18.2 Helping pkgmk locate package contents

The following list describes situations that might require supplying *pkgmk* with extra information and an explanation of how to do so:

- Your development area is not structured in the same way that you want your package structured.

  You should use the *path1*=*path2* pathname format in your *prototype* file.
- You have relocatable objects in your package.

  You can use the *path1*=*path2* pathname format in your *prototype* file, with *path1* as a relocatable name and *path2* a full pathname to that object on your machine.

  You can use the *search* command in your *prototype* file to tell *pkgmk* where to look for objects.

  You can use the *–b basedir* option of *pkgmk* to define a pathname to prepend to relocatable object names while creating the package. For example, executing

  pkgmk –d symbolic-name –b usr2/myhome/reloc

  would look in the directory */usr2/myhome/reloc* for any relocatable object in your package.
- You have variable object names.

  You can use the *search* command in your *prototype* file to tell *pkgmk* where to look for objects.

  You can use the *param*="*value*" command in your *prototype* file to give *pkgmk* a value to use for the object name variables as it creates your package.

  You can use the *variable*=*value* option on the *pkgmk* command line to define a temporary value for variable names.
- The root directory on your machine differs from the root directory described in the *prototype* file (and that

will be used on the installation machine).

You can use the *–r rootpath* option to tell *pkgmk* to ignore the destination pathnames in the *prototype* file. Instead, *pkgmk* prepends *rootpath* to the source pathnames in order to find objects on your machine.

4.1.19 Creating a package with pkgtrans

pkgtrans performs the following package translations:

- a fixed directory structure to a datastream
- a datastream to a fixed directory structure

To perform one of these translations, execute

pkgtrans device1 device2 [pkg1[,pkg2[...]]]

where *device1* is the name of the device where the package currently resides, *device2* is the name of the device onto which the translated package will be placed, and *pkg1(pkg2 ...)* is one or more package names. If no package names are given, all packages residing in *device1* will be translated and placed on *device2*.

If more than one instance of a package resides on *device1*, you must use an instance identifier for *pkg*.

4.1.19.1 Creating a datastream package

Creating a datastream package requires two steps:

1. Create a package using *pkgmk*.

   Use the default device (the installation spool directory) or name a directory into which the package should be placed. *pkgmk* creates a package in a fixed directory format. Specify the capacity of the device where the datastream will be placed as an argument to the *-L* option.
2. After the software is formatted in fixed directory format and is residing in a spool directory, execute *pkgtrans*.

   This command translates the fixed directory format to the datastream format and places the datastream on the specified medium.

For example, the two steps shown below will create a datastream package.

1. pkgmk –d spooldir -l 1400

   formats a package into a fixed directory structure and places it in a directory named *spooldir*. Each part of the package will require no more than 1400 blocks.
2. pkgtrans spooldir 9track package1

   translates the fixed directory format of *package1* residing in the directory *spooldir* into a datastream format. Places the datastream package on the medium in a device named *9track*.

   OR
3. pkgtrans -s spooldir symbolic-name package1

   similar to number 2 above, except that it places the datastream package on the medium in a device named *symbolic-name*. *pkgtrans* will prompt for additional volumes if the package requires more than one diskette.

4.1.19.2 Translating a package instance

When an instance of the package being translated already exists on *device2*, *pkgtrans* will not perform the translation. You can use the *–o* option to tell *pkgtrans* to overwrite any existing instances on the destination device and the *–n* option to tell it to create a new instance if one already exists. Note that this check does not apply when *device2* contains a datastream format.

4.1.20 Quick reference to packaging procedures

Before beginning any packaging procedure, you must first have planned your packaging needs based on the information presented in this chapter. The section entitled Basic steps of packaging" gives a comprehensive list of possible packaging steps and considerations. This section only covers the required steps.

Create a prototype file

1. Create one manually using any editor. There must be one entry for every package component. The format for a *prototype* file entry is:

[volno] ftype class pathname [major minor] [mode owner group]

volno
: designates the medium volume number on which the object should be placed. If no *volno* is given, *pkgmk* distributes package components across volumes automatically.

ftype
: must be one of these object file types:

f
: standard executable or data file

e
: file to be edited upon installation or removal

v
: volatile file, contents will change

d
: directory

x
: exclusive directory

l
: linked file

p
: named pipe

c
: character special device

b
: block special device

i
: installation script or package information file

s
: symbolic link

class
: defines the class to which the object belongs. Place an object into the class of *none* if no special handling is required.

pathname
: defines the pathname of an object. It can be in one of these formats:
  - fixed pathname: */src/myfile*
  - collectively relocatable pathname: *src/myfile* (no beginning slash)
  - individually relocatable pathname: *$BIN/myfile*

  This pathname defines where the component should reside on the installation medium and also tells *pkgmk* where to find it on your machine. If these names differ, use the *path1*=*path2* format for *pathname*, where *path1* is the name it should have on the installation machine and *path2* is the name it has on your machine.

major minor
: defines the major and minor numbers for a block or character special device.

mode owner group
: defines the mode, owner and group for the object. If not defined, the value of the default command is used. If no default value is defined, *644 root other* is assigned.

You can use four types of command lines in a *prototype* file:

- search pathnames

  defines a search path for *pkgmk* to use when creating your package
- include filename

  nests *prototype* files
- default mode owner group

  defines a default *mode owner group* for objects defined in this *prototype* file
- param=value

  defines parameter values for *pkgmk*

All command lines must begin with an exclamation point (*!*).

1. Create one using *pkgproto*.

   pkgproto [–i] [–c class] [path1[=path2] ...] > filename

   where *–i* tells *pkgproto* to record symbolic links with an *ftype* of *f* (not *s*), *–c* defines the class of all objects as *class*, and *path1* defines the object pathname (or names) to be included in the *prototype* file. If *path1* is a directory, entries for all objects in that directory will be generated.

   Use the *path1*=*path2* format to give an object a different pathname in the *prototype* file than it has on your machine. *path1* is the pathname where objects can be located on your machine and *path2* is the pathname that should be substituted for those objects.

   pkgproto writes its output to the standard output. To create a file, you should redirect the output to a file. That file can be named *prototype* (although it is not required).

Create a pkginfo file

Use any editor. Define one entry per line per parameter in this format:

PARAM="value"

where *PARAM* is the name of one of the standard installation parameters defined in the manual entry pkginfo - package characteristics file in Chapter "Reference pages", and value is the value you assign to it.

You can also define values for your own installation parameters using the same format. Names for parameters that you create must begin with a capital letter and be followed by only lower-case letters.

The following five parameters are required in every *pkginfo* file: *PKG*, *NAME*, *ARCH*, *VERSION* and *CATEGORY*. No other restrictions apply concerning which parameters or how many parameters you define.

The *CLASSES* parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the *none* class must be defined in the *CLASSES* parameter before objects belonging to that class will be installed.

Execute pkgmk

pkgmk [–d symbolic-name] [–r rootpath] [–b basedir] [–f filename]

where *–d* specifies that the package should be copied onto *symbolic-name*, *–r* requests that the root directory *rootpath* be used to locate objects on your machine, *–b* requests that *basedir* be prepended to relocatable paths when searching for them on your machine, and *–f* names a file, *filename*, to be used as your *prototype* file. (Other options are described in the manual entry pkgmk - produce an installable package in Chapter "Reference pages".)

Refer to the procedures in this chapter for details on other, optional packaging steps (including how to use *pkgtrans* to create a package in datastream structure).

4.2 Modifying the sysadm interface

Reliant UNIX 5.43 provides a menu interface to the most common administrative procedures. It is invoked by executing *sysadm* and so is referred to as the *sysadm* interface. (A complete description of this interface and instructions on how to use it can be found in [8[).

You can deliver additions or changes to this interface as part of your application software package. Creating

the necessary information for an interface modification is a simple process due to the tools provided by Reliant UNIX 5.43.

This section describes these tools, provides all of the needed background information, and details the procedures necessary to design and write your package administration and to package it so that it will become a part of the administration interface on the installation machine.

The following subsections assume you are familiar with the material covered in Section "Packaging application software".

4.2.1 Introduction to the tools

Two commands can be used to create the files necessary to deliver modifications to the *sysadm* interface as a part of your package.

- *edsysadm* creates all of the files needed for your interface modifications to be installed along with your package
- *delsysadm* deletes menus or tasks from the interface

This section also provides an overview of a group of tools known as the data validation tools. You can use them when writing your system administration to simplify and standardize the programming of administrative interaction. The tools are described in detail in Section "Data validation tools".

The edsysadm command

edsysadm, which allows you to make changes or additions to the interface, is an interactive command that functions much like the *sysadm* command itself. It presents a series of prompts for information. (Which prompt appears depends on your response to the previous prompt.)

After you have responded to all the prompts, *edsysadm* presents a form that you must fill in with information describing the menu or task being changed or added. This form is called the menu (or task) definition form. If you are changing an existing menu or task entry, the definition form will already be filled in with the current values, which you can edit. If you are adding a new menu or task entry, the form will be empty and you will have to fill it in.

When you follow the procedures in this section, *edsysadm* creates all of the files and directories necessary to deliver your interface modifications as a part of your package. The three files that *edsysadm* creates are described in Section "Introduction to the package modification files".

edsysadm builds the directory structure required by the *sysadm* interface. You do not need to know this structure and you are not required to have your work directory organized in any predefined way. When you fill in a menu or task definition form, you supply filenames (for example, a file containing help messages) that *edsysadm* should use when creating the packaging for your interface modifications. *edsysadm* creates a *prototype* file and builds the interface directory format by using the *path1*=*path2* naming convention. *path2* defines where the files reside on your machine and *path1* defines where they should be placed on the installation machine.

The delsysadm command

delsysadm removes tasks and menus from the interface. When you deliver your modifications as a part of your package, you do not need to use *delsysadm* to remove them. Any time an interface modification is delivered as a part of a package, those modifications are automatically removed at the same time as the package. This section describes the *delsysadm* command in case you need to use it on your own machine, for example to remove modifications added for testing.

delsysadm checks for dependencies on the entry being removed before deleting the entry. (A dependency exists if the menu being removed contains an entry placed there by an application package.) If *delsysadm* discovers a dependency, you are asked whether you want to continue with the removal. (If a dependency is found during an automatic removal, the interface entry is not removed.)

When you delete a menu entry with *delsysadm*, it must already be empty (contain no other menus or tasks) or you can execute *delsysadm* with the *–r* option. This option removes a menu and all its entries at the same time.


Use *delsysadm* to remove only those menu or task entries that you have added to an interface.

The data validation tools

The data validation routines help standardize administration interaction in the Version 5.41 environment and also make development easier. The tools are available as shell commands and as visual modules to be used in an FMLI (Form and Menu Language Interpreter) form. The tools perform the following series of tasks:

- prompt a user for a particular type of input
- validate the response
- format and print help and error messages
- return the input if it passes validation

The type of validation performed is defined by the tool itself. For example, the shell command *ckyorn* prompts for and validates an affirmative or negative response. These tools should be used in your administration programs if they are to be added to the *sysadm* interface to maintain consistency within the interface. Refer to Section "Data validation tools", for full details on these tools and their uses.

4.2.2 Introduction to the package modification files

When you execute *edsysadm* to define menus and tasks and save those definitions to be included in your application software package, it creates three files:

1. the package description file
2. the menu information file
3. the *prototype* file

The package description file contains information *edsysadm* uses to change interface modifications already saved for packaging. When you decide to change your modifications after already creating the packaging (meaning the menu information and *prototype* files are already created), the package description file provides *edsysadm* with the information it needs to locate the other package modification files and to make the changes. Without this file, *edsysadm* cannot make such a change. You are asked to supply a name for this file during the *edsysadm* interaction and it is created in your current working directory (unless you supply a full pathname to a different directory with the name).

The menu information file contains the menu or task name, where it is located in the interface structure, and, for tasks, what executable to use when the task is invoked. It tells the interface installation software how to modify the interface structures to include the new definitions. The file's name is the hour, minute, second, day-of-year, and year that the file was created, followed by an *.mi* suffix. It is created in your current working directory.

The *prototype* file created by *edsysadm* contains entries for all of the interface modification components that must be packaged with your software (for example, the menu information file and, for tasks, the executables). These entries must be incorporated into your package either by reading the *edsysadm*-created file into your package *prototype* file or by using the *include* command in the main *prototype* file for your package. The *prototype* file created by *edsysadm* is created in your current working directory with the name of *prototype*.

4.2.3 Overview of the interface modification process

You must take a number of steps to add your package administration to the *sysadm* interface. This section explains each step in detail. The following steps are covered:

- planning your package administration (with details on how to decide if you should modify the interface and where to place it in the interface structure)
- writing your administration actions (with general information on what your executables can be)
- writing your help message (with a description of the required help message file)
- packaging your interface modifications (with procedural details on executing *edsysadm* and what steps must be taken afterwards)

This section also includes instructions on executing *delsysadm*.

4.2.4 Planning your interface modifications

You will need to plan your interface modifications before executing *edsysadm*. Planning begins with deciding if your administration tasks should become a part of the *sysadm* interface. If so, you must decide on where your tasks fit into the interface, what to name your tasks, and the full menu structure involved with your administrative tasks.

4.2.4.1 Deciding if you should modify the interface

Any type of task can be added to the *sysadm* interface with the following two restrictions:

- Tasks that can be automated should not be added to the interface (for example, procedures that can run automatically as part of system booting or as part of your package installation).
- Tasks that require the system to be in firmware mode can be added to the interface but it is strongly recommended that they not be.

Once you have decided to add your administration tasks to the interface, you must determine where in the interface you want to locate tasks and menus.

4.2.4.2 Planning the location of your modifications

To plan your modification you must first become familiar with the interface organization. Then you must decide how to organize the tasks you want to add and how to fit your modifications into the overall structure.

An overview of the interface structure

The *sysadm* interface consists of a hierarchy of menus. At the top of the hierarchy is the main menu (labeled "System Administration Menu"). It appears on the screen, immediately after *sysadm* is invoked, as follows:

```
□□□□□□□□□□□System□Administration□Menu
applications□□□□□-□□Administration□for□Available□Applications
file_systems□□□□□-□□File□System□Creation,□Checking,□and□Mounting
logs□□□□□□□□□□□□□-□□Diagnosing□System□Problems
machine□□□□□□□□□□-□□Machine□Configuration□Display□and□Powerdown
networks□□□□□□□□□-□□Network□Administration
software□□□□□□□□□-□□Software□Installation□and□Removal
storage_devices□□-□□Storage□Device□Operations□and□Definitions
system_setup□□□□□-□□System□Name,□Date/Time□and□Initial□Password□Setup
users□□□□□□□□□□□□-□□User□Login□and□Group□Administration
```

The menus which are supplied are machine-dependent and will not appear on all systems. The applications menu will not appear on the main *sysadm* menu until at least one menu or task has been placed under it. There may be additional menu items on the interface as a result of other products being added.

The main menu consists of a list of function-specific menus. The lefthand column notes the menu names (such as *machine*) and the righthand column gives descriptions of these menus. Each menu offers other menus and/or names of tasks.

```
□□□□□□□□□□□□□□□□□Machine□Management
configuration□-□System□Configuration□Display
shutdown□□□□□□-□Stops□All□Running□Programs□and□Turns□Off□Machine
reboot□□□□□□□□-□Stops□All□Running□Programs□and□Reboots□Machine
whos□on□□□□□□□-□Displays□List□of□Users□Logged□onto□Machine
```

Choosing the entry *configuration* from this screen will cause another menu to be presented. Choosing a task entry, such as *shutdown*, will begin execution of that task.

Planning your administration structure

Planning your administration structure requires three steps:

1. Deciding what tasks to add to the interface.

   You can add any number of tasks. You should have separate entries for each task to be performed. For example, if your administration allowed a log to be changed, added to, and removed, you should create an entry for each task and not combine them into one entry called *log administration*.

2. Deciding under which menu the tasks should be placed.

   You can create new *sysadm* menus at any level and you can change or add to any of the original *sysadm* menus. You should be aware, however, that if you make changes to original menus you might cause problems in the execution of standard *sysadm* operations. It is therefore recommended (though not mandatory) that you create new menus for your package administration by placing it under the *applications* menu (located on the main menu) or by creating a new main menu entry.

3. Organizing your tasks.

   You can organize your tasks under one menu or place them in submenu groups. For example, if your package has tasks to be performed daily and weekly, you might create a structure such as the following:

   - Under the *applications* menu on the *main* menu, add an entry for your package called *pkgAadmin*.
   - Under *pkgAadmin*, add two submenus called *daily* and *weekly*.
   - Under the submenu *daily*, add entries for each of the daily tasks.
   - Under the submenu *weekly*, add entries for each of the weekly tasks.

It is important that you have your full administrative structure planned before running *edsysadm* because you must create a menu entry before placing a task or submenu under it.

After you have planned your structure, you should decide on the names for your menus and tasks.

4.2.4.3 Naming your interface modifications

Naming your interface modifications requires the following three pieces of information described below. This section also details the interface naming requirements and tells you how the system handles naming collisions.

How to name your modifications

When naming your interface modifications, you must decide on these three pieces of information:

Name
: The name of the menu or task as it will appear in the lefthand column of the screen.

Description
: The description of the menu or task as it will appear in the righthand column of the screen.

Location
: The location of a menu or task in the *sysadm* menu hierarchy. This location is a combination, step-by-step, of all the menu names that must be chosen to reach the menu or task. Each step must already exist when the entry is added. For example, when you add a task with a location of *main:applications:mypkg*, you must already have created an entry for the menu *mypkg*.

  All locations begin with *main*. When defining a location in the procedures that follow, each step should be separated by a colon. For example, the *shutdown* task is under the menu *machine*, which, in turn, is under the *main* menu. Thus, the location of the *shutdown* task is *main:machine*.

You will supply these pieces of information on the menu (or task) definition form.

Interface naming requirements

A menu or task name should be as short as possible in length but, at the same time, be descriptive. It can contain only lower case letters and underscores and has a maximum length of 16 characters.

The description field can contain any character string and has a maximum length of 58 characters. This description field text for a menu is also used as the title for that menu when it is displayed. Use of standard title capitalization rules is recommended.

How the system handles naming collisions

A naming collision might occur under two circumstances:

- When the package being installed is an update to an existing version.

  The administrator will be asked during installation if this is an update, in which case the existing menus and tasks will be overwritten.

- When two packages have created identical interface modifications.

  The colliding menu or task will be renamed by adding the first available numerical suffix (beginning with 2). For example, if an entry for *menuA* already exists and a package attempts to add an identical entry, the one being added will be renamed to *menuA2*.

4.2.5 Writing your administration actions

When you execute *edsysadm* to create packaging for a task entry, you will fill in a task definition form. One of the fields on that form asks for the name of the task action file. The task action file is the executable that will run when your task is selected from the interface. Your administrative task can use more than one executable, but, if so, you must create one that is called when the task is selected and call any other executables associated with the task from within it.

The task action can be one of two types:

- Non-interactive

  A non-interactive task action can be any shell executable.

- Interactive

  An interactive task action must be an FMLI form. (Refer to [15] for instructions on writing an FMLI form.)

Use the tools described in Section "Data validation tools", whenever possible when writing administrator interaction.

4.2.6 Writing your help messages

You must write help messages to be packaged with every interface modifications. They are delivered in what is called an *item help file*. This file has text for two types of messages:

1. the help message that will be shown when the user requests help from the parent menu
2. the help messages that will be shown for each field when your task action is an FMLI form

The format of the item help file allows you to create one item help file for each task, combine all of your help messages for multiple tasks into one file, use the same message for multiple FMLI forms, and to define a title hierarchy for the help message screens.

4.2.6.1 The item help file

There are no naming restrictions for the item help file that resides on your machine. However, within the interface structure, the item help file must always be named *Help*. You can use this name if you want to but it is not mandatory since *edsysadm* uses the *path1=path2* naming convention in the *prototype* file that it creates to define the directory structure required by the interface. Regardless of what the item help file is named on your machine, *path1* in the *prototype* file will have the name *Help*. This means that you can have more than one item help file in your working directory at the same time and *edsysadm* will handle the details of giving it the correct name.

There are three types of entries in an item help file:

1. the menu item help
2. the default title (can define both a global default and a form default)
3. the field item help

A description of each type of entry and its format follows. All of the entries use the colon (*:*) as the keyword delimiter.

The menu item help message format

The menu item help message will be shown whenever a user requests help on an entry from the parent menu. Menu item help must be written for each menu and task entry being delivered as an interface modification. For example, if your package administration is adding a menu under main:applications and that menu has three tasks under it, you will need to deliver four menu item help messages.

The format for the menu item help definition is as follows:

[task_name:]ABSTRACT:

□□□□<TAB>□□□Line□1□of□message□text
□□□□<TAB>□□□Line□2□of□message□text
□□□□<TAB>□□□Line□n□of□message□text

<TAB> is a mandatory part of the syntax.

task_name defines the task (or menu) entry to which this help message belongs. This name must match the name that you have decided should appear in the lefthand column of the menu screen. (Refer back to Section "Naming your interface modifications", for more details on this name.) *task_name* is not optional when more than one menu item help definition is defined in the same item help file. This helps to distinguish to which task or menu the message belongs.

The message text should be entered beneath the header line. There can be multiple lines of text with a maximum length of 69 characters per line. Each line must begin with a tab character. Blank lines may be included within the message as long as they also begin with a tab character. An example menu item help definition is shown below.

task1:ABSTRACT:
□□□□□□This□is□line□one□of□the□menu□item□help□message.
□□□□□□This□is□a□second□line□of□message□text.
□□□□□□The□preceding□line□will□appear□as□a□blank□line
□□□□□□when□the□help□message□is□shown□because□it□begins
□□□□□□with□a□tab.

The title for a menu item help message is always the description text, as it appears in the lefthand column of the menu display, prepended by the string *Help on*.

The default title format

You can define two types of default titles:

1. a global default title to be used on all of the help messages defined in the item help file
2. a form default title to be used on all of the help messages defined for a particular form in an item help file with messages defined for numerous forms

Defaults can be overridden, as described in Section "The title hierarchy". A default title definition is recommended but not required.

The format for the default title definition is as follows:

[form_id:]TITLE:Title Text

form_id is the name of the form as it is defined with *lininfo* in your FMLI form definition. When a *form_id* is supplied, this line defines a form default title. When it is not supplied, this line defines a global default title.

The title text defined after the *TITLE* keyword will have the string *HELP on* prepended to it when displayed. Keep this in mind when writing the title.

An example form default title definition is shown below.

task1:TITLE:Package Administration Task1

If *task1* had not been added before *TITLE*, this example would be defining a global default title. The title defined by the example above will be displayed as:

HELP on Package Administration Task1

The field item help message format

The field item help message will be shown whenever a user requests help from within an FMLI form. Each field on the form must have a help message defined in the item help file. The format for the field item help definition is as follows:

[form_id:]field_id:[Title□Text]
□□□□□<TAB>□□□Line□1□of□message□text
□□□□□<TAB>□□□Line□2□of□message□text
□□□□□<TAB>□□□Line□n□of□message□text

form_id is the name of the form as it is defined with *lininfo* in your FMLI form definition. When one item help file contains messages for multiple tasks (and so multiple forms), it is used to distinguish with which form a field belongs. It is optional if the file contains messages for only one task. *field_id* is the name of the field as it is defined with *lininfo* in your FMLI form definition. *Title Text* defines a title used only with the help message for this field. As with the default title, the text defined here will have the string *HELP on* prepended to it when displayed.

The message text should be entered beneath the header line. There can be multiple lines of text with a maximum length of 69 characters per line. Each line must begin with a tab character. Blank lines may be included within the message as long as they also begin with a tab character. An example field item help definition is shown below.

```
task1:fld1:the Name Field
	This is the text for a field item help for a name
	field.
	The preceding line will appear as a blank line
	when the help message is shown because it begins
	with a tab.
```

The title for this field item help message, as defined above, will be *HELP on the Name Field*.

4.2.6.2 The title hierarchy

You can define a global default title, a form default title, and a field title in the item help file. When all three are defined in the same file, the following rules are followed:

- The global default title is used for any message defined in an item help file that does not have a form default title or field title.
- The form default title is used for any message defined in an item help file and that is associated with the form, unless it has a field title.
- The field title is used only for the one field item help message for which it is defined.

In summary, if no field title is defined, the form default title is used. If no form default title is defined, the global default title is used. You always want at least a global default title defined; otherwise, the string *HELP on* will be displayed with no descriptive text.

To define a global default title, add a line to your item help file in the following format:

```
TITLE:Title Text
```

where *Title Text* is the text for the global default title.

To define a form default title, add a line to your item help file in the following format:

```
form_id:TITLE:Title Text
```

where *form_id* is the name of form as it is defined with *lininfo* in your FMLI form definition and *Title Text* is the text for the form default title.

To define a field title, use the following format for the field item help header line:

```
form_id:field_id:Title Text
```

where *form_id* is the name of the form as it is defined with lininfo in your FMLI form definition, *field_id* is the name of the field as it is defined with *lininfo* in your FMLI form definition and *Title Text* is the text for the field title.

In all cases, the text defined as *Title Text* is always prepended with the string *HELP on* when displayed to a user.

4.2.6.3 Setting up for item help in an FMLI object

To help the interface read your item help file and know with which forms and fields a help message is associated, you must define your *help* and *lininfo* descriptors in your FMLI object definition as follows:

- The *help* descriptor must be defined exactly as shown on the line below:

  ```
  help=OPEN TEXT $INTFBASE/Text.itemhelp $LININFO
  ```

- The *lininfo* descriptor for each field must be defined as

  lininfo=[form_id:]field_id

  where *form_id* and *field_id* are names each no longer than 30 characters. The names defined here as *form_id* and *field_id* must match exactly those used as *form_id* and *field_id* in the item help file.

Since you do not create an FMLI form definition for a menu entry, you do not need to take any setup actions. However, you should be certain that the *task_name* keyword precedes the *ABSTRACT* heading line for a menu entry help message.

4.2.6.4 Example item help files

This section shows two example item help files. This figure shows an item help file that defines messages for only one form:

```
ABSTRACT:
□□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□□users□when□they□request□help□while
□□□□□□□□□□viewing□the□parent□menu□for□this
□□□□□□□□□□ask.□□The□task□name□is□"adding□users."
TITLE:Adding□Users
field1:
□□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□□users□when□they□request□help□from□the
□□□□□□□□□□form□and□the□cursor□is□positioned□at
□□□□□□□□□□field1.□□The□title□for□this□message□will
□□□□□□□□□□be□"HELP□on□Adding□Users"□as□defined□above.
field2:Field□2
□□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□□users□when□they□request□help□from□the
□□□□□□□□□□form□and□the□cursor□is□positioned□at
□□□□□□□□□□field2.□□The□title□for□this□message□will
□□□□□□□□□□be□"HELP□on□Field□2".
Note:□□The□lininfo□descriptors□in□the□form□definition□associated□with□this□file
□□□□□□□should□look□like□this:
□□□□□.
□□□□□.
□□□□□.
lininfo=field1
□□□□□.
□□□□□.
□□□□□.
lininfo=field2
```

This figure shows an example of defining messages for multiple forms in one item help file:

```
add:ABSTRACT:
□□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□□users□when□they□request□help□while
□□□□□□□□□□viewing□the□parent□menu□for□the□task
□□□□□□□□□□named□add.
add_user:TITLE:Adding□Users
add_user:field1:
□□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□□users□when□they□request□help□from□the
□□□□□□□□□□form□and□the□cursor□is□positioned□at
□□□□□□□□□□field1.□□The□title□for□this□message□will
□□□□□□□□□□be□"HELP□on□Adding□Users"□as□defined□above.
add_user:field2:Field□2
□□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□□users□when□they□request□help□from□the
□□□□□□□□□□form□and□the□cursor□is□positioned□at
□□□□□□□□□□field2.□□The□title□for□this□message□will
□□□□□□□□□□be□"HELP□on□Field□2".
delete:ABSTRACT:
□□□□□□□□□□The□text□defined□here□will□be□shown□to
```

□□□□□□□□□users□when□they□request□help□while
□□□□□□□□□viewing□the□parent□menu□for□the□task
□□□□□□□□□named□delete.
delete_user:TITLE:Deleting□Users
delete_user:field1:
□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□users□when□they□request□help□from□the
□□□□□□□□□form□and□the□cursor□is□positioned□at
□□□□□□□□□field1.□□The□title□for□this□message□will
□□□□□□□□□be□"HELP□on□Deleting□Users"□as□defined□above.
delete_user:field2:Field□2
□□□□□□□□□The□text□defined□here□will□be□shown□to
□□□□□□□□□users□when□they□request□help□from□the
□□□□□□□□□form□and□the□cursor□is□positioned□at
□□□□□□□□□field2.□□The□title□for□this□message□will
□□□□□□□□□be□"HELP□on□Field□2".
Note:□□The□lininfo□descriptors□in□the□form□definition□associated□with□this
□□□□□□□file□should□look□like□this
□□□□.
□□□□.
lininfo=add_user:field1
□□□□.
□□□□.
lininfo=add_user:field2
□□□□.
□□□□.
lininfo=delete_user:field1
□□□□.
□□□□.
lininfo=delete_user:field2

4.2.7 Packaging your interface modifications

To prepare your interface modifications for installation, you must create the packaging for your menus and tasks by executing *edsysadm*. The packaging created by *edsysadm* consists of two files, a *prototype* file and a menu information file. This section describes the procedures for creating these files and what to do after they have been created. (It also describes how to change the packaging after it has been created.)

edsysadm also creates a package description file. *edsysadm* uses this file during its execution and is not a part of the packaging.

4.2.7.1 Basic steps for packaging your modifications

The procedures described next must be repeated for each menu and task entry being added. Begin with creating the menu entry (or entries) because you cannot add tasks or submenus to a menu that does not exist. Be certain that you use the same package description file name for all of the entries belonging to a package.

After running *edsysadm*, be certain to follow the steps described in Section "Preparing your package", to incorporate the modifications into your software package.

For example, if your administration requires the addition of one menu and four tasks, you will need to follow the procedure for creating the packaging for a menu entry, then repeat the procedure for creating the packaging for a task entry four times. Each time, when asked for a package description file name, give the same name to ensure that the packaging created contains all the necessary entries. These procedures will create a menu information file and a *prototype* file with all of the information necessary to include your interface modifications in your package. The two remaining steps (described in Section "Preparing your package") are to include the *edsysadm* created *prototype* file in your package *prototype* file and to edit the *CLASSES* parameter in the *pkginfo* file.

4.2.7.2 Creating or changing the packaging for a menu entry

The procedures for creating and changing the packaging for a new menu are similar and both result in the display of a menu definition form. Each procedure is described below, followed by a description of the menu

definition form.

Creating the packaging for a menu entry

Before creating the packaging for a new menu entry, you should:

- Select a name and description for the menu.
- Select a location for it in the interface.
- Prepare a help message file for the menu entry (refer to Section "Writing your help messages", for instructions).
- Know the name of the package description file to which the information for this menu should be added (if you are adding multiple menus and tasks)

If you do not execute the following command from the directory in which the help message file resides, supply the full pathname when prompted for the name of the help message file.

1. Type *edsysadm* and press <RETURN>.
2. You are asked to choose between a menu and a task. Choose *menu* and press <RETURN>.
3. You are asked to choose between adding a new menu or changing an existing one. Choose *add* and press <RETURN>.
4. You are given an empty menu definition form. Fill it in and press <SAVE>. (See the The menu definition form for descriptions of the fields on this form.)
5. You are asked if you want to test the changes before actually making them. Answer either *yes* or *no* and press <SAVE>. (If you answer *yes*, refer to the section entitled Testing your menu changes on-line to learn what the test involves.)
6. You are asked if you want to install the modifications into the interface on your machine or save them for a package. Choose *save* and press <SAVE>.
7. You are asked to supply a file name. Enter a name for the package description file and press <SAVE>.
8. If the file name given for the package description file already exists, you are asked if you want to overwrite it or add to its contents. Answer *overwrite*, *do not overwrite*, or *add* and press <SAVE>.
9. If the file name does not already exist (or after you have completed Step 8) you will see a message stating that the menu information file and *prototype* file have been verified and the top-level *prototype* must be edited to include the new *prototype* file. Press <CANCEL> to return to the menu shown in step 3. Press <CONT> to return to the form shown in step 4.

Changing the packaging for a menu entry

Before changing the packaging for a menu entry, you should:

- Know the name and description of the menu entry.
- Know its location in the interface.
- Change the associated help message file, if necessary, or create a new one (refer to Section "Writing your help messages", for instructions).
- Know the name of the package description file associated with the package being changed (and know that it is available in your current working directory).

If you have changed a help message file or created a new one and you do not execute the following command from the directory in which the help message file resides, supply the full pathname when asked for the name of the file.

1. Type *edsysadm* and press <RETURN>.
2. You are asked to choose between a menu and a task. Choose *menu* and press <RETURN>.
3. You are asked to choose between adding a new menu and changing an existing one. Choose *change* and press <RETURN>.
4. You are asked if your change is for an on-line menu or for a menu that has been saved for a package. Choose *packaged* and press <SAVE>.

5. You are asked to supply the package description file name for the package being changed. Fill in the name of a valid package description file and press <SAVE>.
6. You are given a menu definition form filled in with the current values for the menu named above. Make the desired changes and press <SAVE>. (See the The menu definition form for descriptions of the fields on this form.)
7. You are asked if you want to test the changes before actually making them. Answer either *yes* or *no* and press <SAVE>. (If you answer *yes*, refer to the section entitled Testing your menu changes on-line to learn what the test involves.)
8. You are asked if you want to install the modifications into the interface on your machine or save them for a package. Choose *save* and press <SAVE>.
9. You are asked to supply a file name. Enter a name for the package description file and press <SAVE>. (This must be the same package description file named in Step 5.)
10. If the file name given for the package description file already exists, you are asked if you want to overwrite it or add to its contents. Answer *overwrite*, *do not overwrite*, or *add* and press <SAVE>.
11. If the file name does not already exist (or after you have completed Step 10) you will see a message stating that the menu information file and *prototype* file have been verified and the top-level *prototype* must be edited to include the new *prototype* file. Press <CANCEL> to return to the menu shown in step 4 where you selected *packaged*. Press <CONT> to return to the form shown in step 5 where you entered the name of a valid package description file.

Testing your menu changes on-line

Before installing your menu changes, you may want to verify that you've added an entry to a menu. The *edsysadm* command gives you a chance to do this after you fill in the menu definition form. Follow these steps to perform your test.

1. Type *yes* when *edsysadm* presents the following prompt:

   Do you want to test this modification before continuing?
2. The parent menu (on which your addition or change is listed) is displayed. Check to make sure your modification has been made correctly.
3. Put the cursor on the new or changed menu entry and press the <HELP> key. The text of the help message for that menu entry is displayed so you can check it. (Press <CANCEL> to return to the menu.)
4. To exit on-line testing, press the <CANCEL> key.
5. You are returned to the prompt:

   Do you want to test this modification before continuing?

   If you want to continue executing the change, type *no*.

   If you want to make additional modifications to the menu definition form, press <CANCEL>. You are returned to the form and can make further changes at that time. (Press <SAVE> when you have finished your editing. You can then retest your changes or continue executing the change.)

The menu definition form

This form contains four fields in which you must provide: a menu name, a menu description, a menu location, and the name of the help message for the menu. Below are descriptions of the information you must provide in each field.

Menu Name
: The name of the new menu (as it should appear in the lefthand column of the screen). This field has a maximum length of 16 alphanumeric characters.

Menu Description
: A description of the new menu (as it should appear in the righthand column of the screen). This field has a maximum length of 58 characters and can consist of any alphanumeric character except the at sign (@), caret (^), tilde (~), back grave (´), grave ('), and double quotes (").

Menu Location
: The location of the menu in the menu hierarchy, expressed as a menu pathname. The pathname should begin with the main menu followed by all other menus that must be traversed (in the order they are traversed) to access this menu. Each menu name must be separated by colons. For example, the menu location for a menu entry being added to the Applications menu is *main:applications*. *Do not include the menu name in this location definition*. The complete pathname to this menu entry will be the menu location plus the menu name defined at the first prompt.

  This is a scrollable field, showing a maximum of 50 alphanumeric characters at a time.

Menu Help File Name
: Pathname to the item help file for this menu entry. If it resides in the directory from which you invoked *edsysadm*, you do not need to give a full pathname. If you name an item help file that does not exist, you are placed in an editor (as defined by $EDITOR) to create one. The new file is created in the current directory and named *Help*.

The following screen shows a filled-in sample menu definition.

□□□□□□□□□□□□□Define□A□Menu
Name:□□msvr
Description:□□Menu□Description
Location:□□main:applications
Help□Message:□□Help

4.2.7.3 Creating or changing the packaging for a task entry

The procedures for creating and changing the packaging for a new task are similar and both result in the display of a task definition form. Each procedure is described below, followed by a description of the task definition form.

Creating the packaging for a task entry

Before creating the packaging for a task entry, you should:

- Gather all files that will be associated with this task, such as the help file, FMLI forms, or other executables. All files should already be prepared.
- Decide on the task name and description.
- Decide on its location in the interface.
- Create a help file (refer to Section "Writing your help messages", for instructions).
- Know the name of the package description file to which the information for this task should be added (if you are adding multiple menus and tasks)

If you do not execute the following command from the same directory in which the files associated with this task reside, enter full pathnames when supplying file names.

1. Type *edsysadm* and press <RETURN>.
2. You are asked to choose between a menu and a task. Choose *task* and press <RETURN>.
3. You are asked to choose between adding a new task or changing an existing one. Choose *add* and press <RETURN>.
4. You are given an empty task definition form. Fill it in and press <SAVE>. (See The task definition form for descriptions of the fields on this form. Be aware that, when you name the menu under which you want this new task to reside, that menu must already be packaged.)
5. You are asked if you want to install the modifications into the interface on your machine or save them for a package. Choose *save* and press <SAVE>.
6. You are asked to supply a file name. Enter a name for the package description file and press <SAVE>.
7. If the file name given for the package description file already exists, you are asked if you want to overwrite it or add to its contents. Answer either *overwrite*, *do not overwrite*, or *add* and press <SAVE>.
8. If the file name does not already exist (or after you have completed Step 7) you see a message stating that the menu information file and *prototype* file have been verified and the top-level *prototype* must be edited to

include the new *prototype* file. Press <CANCEL> to return to the menu shown in step 3. Press <CONT> to return to the form shown in step 4.

Changing the packaging for a task entry

Before changing the packaging for a task entry, you should:

- Gather any of the files associated with this task that have been changed or are new. All files should already be prepared or changed.
- Know the menu name and description.
- Know its location in the interface.
- Change the associated help file, if necessary (refer to Section "Writing your help messages", for instructions).
- Know the name of the package description file associated with the package being changed (and know that it is available in your current working directory).

If your change requires new files or changes to existing files and you do not execute the following command from the directory in which the files reside, enter full pathnames when supplying file names.

1. Type *edsysadm* and press <RETURN>.
2. You are asked to choose between a menu and a task. Choose *task* and press <RETURN>.
3. You are asked to choose between adding a new task and changing an existing one. Choose *change* and press <RETURN>.
4. You are asked if your change is for an on-line task or for a task that has been saved for a package. Choose *packaged* and press <SAVE>.
5. You are asked to supply the package description file name for the package being changed. Fill in the name of a valid package description file and press <SAVE>.
6. You are given a task definition form filled in with the current values for the task named above. Make the desired changes and press <SAVE>. (See The task definition form for descriptions of the fields on this form.)
7. You are asked if you want to install the modifications into the interface on your machine or save them for a package. Choose *save* and press <SAVE>.
8. You are asked to supply a file name. Enter a name for the package description file and press <SAVE>. (This must be the same package description file named in Step 5.)
9. If the file name given for the package description file already exists, you are asked if you want to overwrite it or add to its contents. Answer either *overwrite*, do not overwrite, or *add* and press <SAVE>.
10. If the file name does not already exist (or after you have completed Step 9) you see a message stating that the menu information file and *prototype* file have been verified and the top-level *prototype* must be edited to include the new *prototype* file. Press <CANCEL> to return to the menu shown in step 4. Press <CONT> to return to the form shown in step 5.

The task definition form

This form contains six fields in which you must provide: a task name, a task description, a task location, the name of a help message for the task, a task action file, and the files associated with the task. Below are descriptions of the information you must provide in each

Task Name
: The name of the new task (as it should appear in the lefthand column of the screen). This field has a maximum length of 16 alphanumeric characters.

Task Description
: A description of the new task (as it should appear in the righthand column of the screen). This field has a maximum length of 58 characters and can consist of any alphanumeric character except the at sign (@), carat (^), tilde (~), back grave (´), grave ('), and double quotes (").

Task Location

The location of the task in the menu hierarchy, expressed as a pathname. The pathname should begin with the main menu followed by all other menus that must be traversed (in the order they are traversed) to access this task. Each menu name must be separated by colons. For example, the task location for a task entry being added to the applications menu is *main:applications*. *Do not include the task name in this location definition*. The complete pathname to this task entry will be the task location as well as the task name defined at the first prompt.

This is a scrollable field, showing a maximum of 50 alphanumeric characters at a time.

Task Help File Name

Pathname to the item help file for this task entry. If it resides in the directory from which you invoked *edsysadm*, you do not need to give a full pathname. If you name an item help file that does not exist, you are placed in an editor (as defined by $EDITOR) to create one. The new file is created in the current directory and named *Help*.

Task Action

The FMLI form name or executable that will be run when this task is selected. This is a scrollable field, showing a maximum of 58 alphanumeric characters at a time. This pathname can be relative to the current directory as well as absolute. (Refer to Section "Writing your administration actions", for details.)

Task Files

Any FMLI objects or other executables that support the task action listed above and might be called from within that action. *Do not include the help file name or the task action in this list*. Pathnames can be relative to the current directory as well as absolute. A dot (.) implies 'all files in the current directory' and includes files in sub-directories. This is a scrollable field, showing a maximum of 50 alphanumeric characters at a time.

The following screen shows a filled-in sample task definition form.

Name:□□msvrtask
Description:□□Task□Description
Location:□□main:applications:msvr
Help□Message:□□Help
Action:□□Form.msvrtask
Task□Files:□□Form.task2,□Text.task2

4.2.7.4 Preparing your package

You must perform two steps, after executing *edsysadm*, to include your interface modification files in your application package.

1. Include the *prototype* file

   The *prototype* file that *edsysadm* creates must become a part of your package *prototype* file structure. This means that you must either read it into another *prototype* file or use the *include* command in your primary *prototype* file. For example, adding

   !include /myproject/admsrc/prototype

   to a *prototype* file in the */myproject* directory ensures that the *prototype* file in */myproject/admsrc*, and all of the objects it describes, will be included when the packaging tool, *pkgmk*, creates the package.

2. Change your *CLASSES* parameter in the *pkginfo* file

   The components defined in the *prototype* file that *edsysadm* creates are placed into the two special classes: *OAMmif* and *OAMadmin*. You must edit the *pkginfo* file for your package and add these to the *CLASSES* parameter definition. For example, a *CLASSES* definition before the change might look like this:

   CLASSES="class1 class2"

   It should be changed to look like this:

   CLASSES="class1 class2 OAMmif OAMadmin"

Your interface modifications are now ready to be included in your package when you create your package using *pkgmk*. (Details on packaging procedures are discussed in Section "Packaging application software".)

4.2.8 Integrating FMLI interfaces in an OA&M interface

To integrate FMLI interfaces in an OA&M interface you need to take the following steps:

1. Create an interface file (usually with a *.mi* suffix). The syntax is:

   menu_path:item^description^[action]

   menu_path always starts with "main". *item* and *description* can be internationalized strings. For example:

   main:$$SIsadmex_cat:303:"performance"^$$SIsadmex_cat:248:"Monitoring□System Activity□and□Workload"^
   main:performance:sar^$$SIadmex_cat:26:"System□Activity□Report"^Menu.sar

   Files identified by *action* are looked for in:

   /usr/sadm/sysadm/add-ons/$PKGINST/menu_path_ohne_main/description

   $PKGINST is replaced by the name of the package. For example, the full path for *Menu.sar* is:

   /usr/sadm/sysadm/add-ons/$PKGINST/performance/sar/Menu.sar

2. Create help files.

   Help files are looked for as appropriate to the *$LANG* specification.

3. Add the interface files to the package (*prototype* file).
   - The *.mi* file must belong to the *OAMmif* class.
   - Other interface files may belong to *OAMadmin* or special classes.

The following figure is an example of a *prototype* file for integrating FMLI interfaces.

x□OAMmif□/var/sadm/pkg/$PKGINST/save/intf_install□0755□root□sys
v□OAMmif□/var/sadm/pkg/$PKGINST/save/intf_instal/performance.mi=$PACKDIR/
□□performance/performance.mi
x□OAMadmin□$OAMBASE/add-ons/$PKGINST/performance□0755□root□sys
x□OAMadmin□$OAMBASE/add-ons/$PKGINST/performance/sar□0755□root□sys
f□OAMadmin□$OAMBASE/add-ons/$PKGINST/performance/sar/Menu.sar
f□OAMadmin□$OAMBASE/add-ons/$PKGINST/performance/Help
f□OAMadmin□$OAMBASE/add-ons/$PKGINST/performance/sar/Help

The help texts for additional languages should be placed in a separate package. The lines for a language-dependent package in the *prototype* file will be something like:

f□OAMadmin□$OAMBASE/add-ons/SIsadmex/performance/$LANG/Help
f□OAMadmin□$OAMBASE/add-ons/SIsadmex/performance/$LANG/sar/Help

The actual entries in the OA&M inteface will be made by the *OAMmif* class action script. Searching for language-dependent help files is handled by FMLI.

4.2.9 Deleting interface modifications

Interface modifications can be deleted in two ways. When a package is removed, the modifications installed with the package are removed automatically. Modifications can also be removed online by executing *delsysadm*.

To delete either a menu or task entry online, execute

delsysadm□□name

where *name* is the location of the task or menu in the interface, followed by the menu or task name. For example, to delete a task named *mytask* with the location *main:application:mymenu*, execute

delsysadm□□main:application:mymenu:mytask

Before an entry for a menu can be removed, that menu must be empty (contain no submenus or tasks). If it is not, you must use the *–r* option with *delsysadm*. This option requests that, in addition to the named menu, all submenus and tasks located under that menu be removed. For example, to remove *main:application:mymenu* and all submenus and tasks that reside under it, execute

delsysadm□□–r□□main:application:mymenu

When you use the *–r* option, *delsysadm* checks for dependencies before removing any subentries. (A dependency exists if the menu being removed contains an entry placed there by an application package.) If a dependency is found, you are shown a list of packages that depend on the menu you want to delete and

asked whether you want to continue. If you answer yes, the menu and all of its menus and tasks are removed (even those shown to have dependencies). If you answer no, the menu is not deleted.

Use *delsysadm* to remove only those menu or task entries that you have added to the interface with *edsysadm*.

4.3 Data validation tools

The data validation tools are a group of shell level commands that serve two purposes:

- standardize the appearance of administration interaction in the Reliant UNIX 5.43 environment regardless of who writes it
- simplify development of scripts requiring administrator input

Every tool generates a prompt, validates the answer and returns the response. There are no restrictions on when you should use them. It is recommended that you use them every time your application interacts with an administrator. Using the tools at such a time will make all administrator interaction look alike to the user, regardless of the vendor who created the package. You will see, as well, that using these tools makes writing scripts with administrator interaction much simpler.

At the very least, it is recommended that you use them in your request script (the packaging script from which you can solicit administrator input) and in the executables you deliver when your package administration will be incorporated into the *sysadm* interface. See Section "Modifying the sysadm interface", for details about writing executables for the *sysadm* interface and Section "Packaging application software", for details on writing a request script.

This section introduces you to the data validation tools and discusses their characteristics. For details on a specific tool, look in Chapter "Reference pages". The shell commands and corresponding visual tools are provided as manual pages.

4.3.1 Types of tool

There are two types of data validation tool. Both perform the same series of tasks (described later) but are used in different environments. The two types are:

1. Shell commands

   These tools are invoked from the shell level and used in shell scripts.

2. Visual tools

   These tools are invoked from within the field definition in an FMLI form definition. While the shell commands perform all tasks with one command, the visual tools are broken into separate commands for defining help messages, error messages and performing validation.

4.3.2 Characteristics of the tools

All of the shell commands perform the same series of tasks (the visual tools each perform a subsection of the full series). Those tasks are:

- Prompt a user for input
- Validate the answer
- Print a help message when requested
- Present an error message when validation fails
- Return the input if it passes validation
- Allow a user to quit the process

The tool itself defines the type of prompt shown and validation performed is defined. For example, the shell command *ckyorn* prompts for a yes or no answer and accepts only a positive or negative response. Other tools expect other types of input. *ckrange* prompts for an integer and checks that it is within a given range. The limits of the range can be passed to the *ckrange* call as a parameter.

Leading and trailing white space is stripped from the input before validation is performed.

4.3.2.1 The data validation tool prompts

The default prompt for each of these commands can be added to or overwritten using the *-p* option of the command. The manual page for each tool (see Chapter "Reference pages") shows the default prompt text.

For example, executing *ckyorn* without options produces the following output:

Yes or No [y,n,?,q]:

The next example shows the use of the *-p* option and the output that is produced.

```
$ ckyorn -p "Do you want the manual page files installed?"
Do you want the manual page files installed? [y,n,?,q]:
```

4.3.2.2 The data validation tool help messages

Each tool has a help message that you can use as is, add to, or completely overwrite. You must use the *-h* option of a shell command to change the default message. The manual page for each tool (see Appendix B) shows the default help message text.

For example, if you executed *ckyorn* without options and the user requested a help message by entering *?* at the prompt, the following message would be seen:

```
To respond in the affirmative, enter y, yes, Y, or YES.
To respond in the negative, enter n, no, N, or NO.
```

The next example shows the use of the *-h* option when executing *ckyorn*. The text defined after the *-h* will be shown if the user requests a help message.

```
ckyorn -h "Answer yes if you want the manual page files \
installed or no if you do not."
```

If you insert a tilde (~) at the beginning or end of your definition, the default text will be added at that point. For example,

```
ckyorn -h "The manual page files will be written to your \
system, or not, based on your answer.~"
```

will produce the help message:

```
The manual page files will be written to your system, or not,
based on your answer. To respond in the affirmative, enter y,
yes, Y, or YES. To respond in the negative, enter n, no, N, \
or NO.
```

4.3.2.3 The data validation tool error messages

Each tool has a default error message that you can use as is, add to, or completely overwrite. You must use the *-e* option of a shell command to change the default. The manual page for each tool (see Chapter "Reference pages") shows the default error message text.

For example, if you executed *ckyorn* without options, and validation failed, the following message would be seen:

ERROR: Please enter yes or no.

The next example shows the use of the *-e* option when executing *ckyorn*. The text defined after the *-e* will be prepended with *ERROR:* and shown if validation fails.

```
ckyorn -e "You did not respond with yes or no."
```

If you insert a tilde (~) at the beginning or end of your definition, the default text will be added at that point.

4.3.2.4 Message formatting

All three message types (prompt, error, and help) are limited in length to 78 characters and are automatically formatted. Regardless of how you define them in your code, any white space used (including newline) is stripped during formatting.

You can use the *-W* option of a shell command (or the *ckwidth* variable of a function) to define the line length to which your messages should be formatted.

4.3.3 The shell commands

The following table lists the commands and what they are used for. All of the shell commands perform the same series of tasks, as described previously. The table's 'Purpose' column describes the type of prompt and validation with which the command deals. Details for each command can be found on the respective manual page in Chapter "Reference pages"."

Command (and function)	Purpose
ckdate	Prompts for and validates that the answer is a date (can define format for date).
ckgid	Prompts for and validates that the answer is a group id.
ckint	Prompts for and validates an integer value (can define base for input).
ckitem	Builds a menu, prompts for and validates a menu item (can define characteristics of the menu).
ckkeywd	Adds keywords to a prompt and validates that the return answer matches a keyword.
ckpath	Prompts for and validates a pathname (can define what type of validation to perform, such as "pathname must be readable").
ckrange	Prompts for and validates an integer within a range (can define the upper and lower limits of the range).
ckstr	Prompts for and validates that the answer is a string (can define a regular expression, in which case the string must match the expression).
cktime	Prompts for and validates that the answer is a time (can define format for time).
ckuid	Prompts for and validates that the answer is a user id.
ckyorn	Prompts for and validates a yes/no answer. Input must be y, yes, Y, YES, n, no, N, or NO.
dispgid	Displays a list of all valid group names.
dispuid	Displays a list of all valid user names.

Table 32: The shell commands

4.3.4 The visual tools

The visual tools are invoked from within the field definition of an FMLI form. Because of the nature of FMLI form definitions, it is necessary to divide the tasks performed by only one shell command into sets. The purpose of a visual tool set parallels the purpose of a shell command. For example, *ckdate* performs a group of tasks for a prompt whose response should be a date. The same group of tasks requires three visual tools:

errdate
: formats and presents an error message

helpdate
: formats and presents a help message

valdate
: validates the answer to be a date

The format and description of each visual tool set is shown on the equivalent shell command manual page in Chapter "Reference pages". Refer to the manual page ckdate, errdate, helpdate, valdate - prompts for and validates a date, for details on the three visual tools *errdate*, *helpdate*, and *valdate*.

The following table lists the visual tool sets and their associated response type.

Visual tool set	Response type
erryorn, helpyorn, valyorn	yes or no
errint, helpint, valint	integer
errange, helprange, valrange	integer in a range
errstr, helpstr, valstr	string (potentially matching an expression)
errpath, helppath, valpath	pathname
erritem, helpitem	menu item
errgid, helpgid, valgid	existing group
errtime, helptime, valtime	time of day
errdate, helpdate, valdate	date

Table 33: The visual tool sets

4.4 Package installation case studies

This section presents packaging case study in order to show packaging techniques such as

Case 1
: installing objects conditionally,

Case 2
: determining at run time how many files to create,

Case 3
: creating a database file at installation time,

Case 4
: using optional packaging files,

Case 5
: how to modify an existing data file during package installation and removal

- using the *sed* class and a postinstall script,
- using class action scripts and creating classes,
- using the *build* class,

Case 6
: using classes and class action scripts.

Each case begins with a description of the study, followed by a list of the packaging techniques it uses and a narrative description of the approach taken when using those techniques. After this material, sample files and scripts associated with the case study are shown.

4.4.1 Case #1: Installing objects conditionally

This package has three types of objects. The installer may choose which of the three types to install and where to locate the objects on the installation machine.

Techniques

This case study shows examples of the following techniques:

- using variables in object pathnames
- using the request script to solicit input from the installer
- setting conditional values for an installation parameter

Approach

To set up selective installation, you must:

- Define a class for each type of object which can be installed.

  In this case study, the three object types are the package executables, the manual pages, and the *emacs* executables. Each type has its own class: bin, man, and emacs, respectively. Notice in the *prototype* file, shown in the figure "Case #1 prototype file" on ..., that all of the object files belong to one of these three classes.

- Initialize the *CLASSES* parameter in the *pkginfo* file as null.

  Normally when you define a class, you want the *CLASSES* parameter to list all classes that will be installed. Otherwise, no objects in that class will be installed. For this example, the parameter is initially set to null. *CLASSES* will be given values by the request script, based on the package pieces chosen by the installer. This way, *CLASSES* is set to only those object types that the installer wants installed. The figure "Case#1 pkginfo file" on ... shows the *pkginfo* file associated with this package. Notice that the *CLASSES* parameter is set to null.

- Define object pathnames in the *prototype* file with variables.

  These variables will be set by the request script to the value which the installer provides. *pkgadd* resolves these variables at installation time and so knows where to install the package.

The three variables used in this example are:

$NCMPBIN
: defines location for object executables

$NCMPMAN
: defines location for manual pages

$EMACS
: defines location for emacs executables

Look at the example *prototype* file (Figure "Case #1 prototype file" on ...) to see how to define the object pathnames with variables.

- Create a request script to ask the installer which parts of the package should be installed and where they should be placed.

  The request script for this package, shown in the figure "Case study #1 request script" on ..., asks two questions:

  1. Should this part of the package be installed?

     When the answer is yes, then the appropriate class name is added to the *CLASSES* parameter. For example, when the question "Should the manual pages associated with this package be installed" is answered yes, the class *man* is added to the *CLASSES* parameter.

  2. If so, where should that part of the package be placed?

     The appropriate variable is given the value of the response to this question. In the manual page example, the variable *$NCMPMAN* is set to this value.

These two questions are repeated for each of the three object types.

At the end of the request script, the parameters are made available to the installation environment for *pkgadd* and any other packaging scripts. In the case of this example, no other scripts are provided.

When looking at the request script for this example, notice that the questions are generated by the data validation tools *ckyorn* and *ckpath*.

Sample Files

Case #1 pkginfo file:

```
PKG='ncmp'
NAME='NCMP Utilities'
CATEGORY='applications,tools'
ARCH='i386'
VERSION='Release 1.0, Issue 1.0'
CLASSES='
```

Case #1 prototype file:

```
i pkginfo
i request
x bin $NCMPBIN 0755 root other
f bin $NCMPBIN/dired=/usr/ncmp/bin/dired 0755 root other
f bin $NCMPBIN/less=/usr/ncmp/bin/less 0755 root other
f bin $NCMPBIN/ttype=/usr/ncmp/bin/ttype 0755 root other
f emacs $NCMPBIN/emacs=/usr/ncmp/bin/emacs 0755 root other
x emacs $EMACS 0755 root other
f emacs $EMACS/ansii=/usr/ncmp/lib/emacs/macros/ansii 0644 root other
f emacs $EMACS/box=/usr/ncmp/lib/emacs/macros/box 0644 root other
f emacs $EMACS/crypt=/usr/ncmp/lib/emacs/macros/crypt 0644 root other
f emacs $EMACS/draw=/usr/ncmp/lib/emacs/macros/draw 0644 root other
f emacs $EMACS/mail=/usr/ncmp/lib/emacs/macros/mail 0644 root other
f emacs $NCMPMAN/man1/emacs.1=/usr/ncmp/man/man1/emacs.1 0644 root other
d man $NCMPMAN 0755 root other
d man $NCMPMAN/man1 0755 root other
f man $NCMPMAN/man1/dired.1=/usr/ncmp/man/man1/dired.1 0644 root other
f man $NCMPMAN/man1/ttype.1=/usr/ncmp/man/man1/ttype.1 0644 root other
f man $NCMPMAN/man1/less.1=/usr/ncmp/man/man1/less.1 0644 inixmr other
```

Case study #1 request script:

```
trap 'exit 3' 15
# determine if and where general executables should be placed
ans=`ckyorn -d y \
-p "Should executables included in this package be installed"
` || exit $?
if [ "$ans" = y ]
then
        CLASSES="$CLASSES bin"
        NCMPBIN=`ckpath -d /usr/ncmp/bin -aoy \
                -p "Where should executables be installed"
        ` || exit $?
fi
# determine if emacs editor should be installed, and if it should
# where should the associated macros be placed
ans=`ckyorn -d y \
        -p "Should emacs editor included in this package be installed"
` || exit $?
if [ "$ans" = y ]
then
        CLASSES="$CLASSES emacs"
        EMACS=`ckpath -d /usr/ncmp/lib/emacs -aoy \
                -p "Where should emacs macros be installed"
        ` || exit $?
fi
# determine if and where manual pages should be installed
ans=`ckyorn \
        -d y \
        -p "Should manual pages associated with this package be installed"
` || exit $?
if [ "$ans" = y ]
then
        CLASSES="$CLASSES man"
```

```
□□□□□□□□NCMPMAN=`ckpath□-d□/usr/ncmp/man□-aoy□\
□□□□□□□□□□□□□□□□□□-p□"Where□should□manual□pages□be□installed"
□□□□□□□□`□||□exit□$?
fi
#□make□parameters□available□to□installation□service,
#□and□so□to□any□other□packaging□scripts
cat□>$1□<<!
CLASSES='$CLASSES'
NCMPBIN='$NCMPBIN'
EMACS='$EMACS'
NCMPMAN='$NCMPMAN'
!
exit□0
```

4.4.2 Case #2: Determining how many files to create

This package installs a driver. A set of device nodes associated with that driver needs to be created, but the installer will decide how many nodes to create. After installation, the system needs to be rebooted so that the driver is properly configured and the device nodes set up.

Techniques

This case study shows examples of the following techniques:

- different organization of product and target environment
- installing a driver with a *postinstall* script
- using an exit code to reboot the system
- allowing the installer to define how many device nodes are to be created after rebooting
- user-definable temporary directories.
- adding the packaged files to the software installation database (*idinstallf*).

Approach

To install the driver, you must:

1. Include the object, system and master files for the driver in the *prototype* file.

   In this example, the object file for the driver is a data file named *xxx.o*. The system and master files are *sdev* and *mdev* respectively. Within the *postinstall* script these files and the node file created during installation are set up as appropriate to the standard driver installation command.

   Looking at figure "Case #2 prototype file" on ..., notice the following:

   - Since no special treatment is required for these files, you can put them into the standard *none* class. The *CLASSES* parameter is set to none in the *pkginfo* file (Figure "Case #2 pkginfo file" on ...).
   - The pathname for the driver modules begins with the variable *$WHERE*. This variable will be set in the request script and allows the administrator to decide where the driver modules should be installed at installation time. The default directory will be */tmp*.
   - There is an entry for the *postinstall* script (which will be executed after the files have been added to the file system).

2. Create a request script.

   The request script, shown in the figure "Case #2 Request Script" on ..., has two major functions:

   - to determine how many device nodes to create for this driver

     This is accomplished by questioning the installer and then assigning the answer to the parameter *$NDEVICES*. Notice that the data validation tool *ckrange* is used and that it limits the response to a number between 0 and 8. It sets the default number to 3.
   - to determine where the installer wants the driver objects to be installed

     This is accomplished by questioning the installer and assigning the answer to the *$WHERE* parameter.

   The script ends with a routine to make the parameters *NDEVICES* and *WHERE* available to the installation

environment and so to the *postinstall* script.

3. Create a *postinstall* script.

   The *postinstall* script, shown in the figure "Case #2 postinstall script" on ..., actually performs the driver module installation. It is executed after the files have been installed. The *postinstall* shown for this example performs the following actions:

   - calculates the minor numbers for installed devices
   - generates the node file
   - executes the *idinstall* command, passing it the name of the driver as a parameter
   - finalizes the installation using *idinstallf –f*
   - deletes any temporary files with *removef*

4. Reboot the system upon installation.

   This is accomplished by exiting from the *postinstall* script with an exit code of 10, meaning that the system should be rebooted upon completing an error-free installation.

Sample files

Case #2 prototype file:

```
i□pkginfo
i□copyright
i□postinstall
i□request
i□preremove
!□default□755□root□other
f□none□$WHERE/drvdemo/xxx.Dr=xxx.o
f□none□$WHERE/drvdemo/xxx.Ma=mdev
f□none□$WHERE/drvdemo/xxx.Sy=sdev
```

Case #2 pkginfo file:

```
PKG=drvdemo
NAME=driver□installation□demonstration□package
ARCH=i386
VENDOR=Siemens□Nixdorf□Informationssysteme□AG
VERSION=1.0
CATEGORY=system
CLASSES=none
```

Case #2 Request Script:

```
trap□'exit□3'□15
#□request□number□of□devices□to□be□created
NDEVICES=`ckrange□-l0□-u7□-d3□\
□□□□□□□□-p□"How□many□devices□do□you□want□configur?"□`□||□exit□$?
#□determine□where□to□put□the□objects□during□installation□temporarily
WHERE=`ckpath□-aoy□-d□/temp□\
□□□□□□□□-p□"Where□do□you□want□driver□objects□to□be□put□during□installation?
□□□□□□□□□□□□□Note:□Directory□must□already□exist!"□`□||□exit□$?
#□make□parameter□available□to□installation□environment
cat□>$1□<<!
NDEVICES=$NDEVICES
WHERE=$WHERE
!□□□□□□□□□□□□□□□□-p□"Where□do□you□want□driver□object□installed"
exit□0
```

Case #2 postinstall script:

```
#□PKGINST□parameter□provided□by□installation□service
#□NDEVICES□parameter□provided□by□'request'□script
#□BOOTDIR□parameter□provided□by□'request'□script
if□[□-f□/tmp/${PKGINST}.debug□]
then
```

```
set -x
fi
# install a module. $1 is the module name
do_install() {
    ERR=${WHERE}/${1}.err
    IDCOMPS="Driver.o Master System Mfsys Sfsys Rc Node Space.c"
    if
        [ -f ${1}.Dr ]
    then
        mv ${1}.Dr Driver.o
    fi
if
        [ -f ${1}.Sp ]
    then
        mv ${1}.Sp Space.c
    fi
    if
        [ -f ${1}.Ma ]
    then
        grep -v "[\*#]ident" ${1}.Ma > Master
        rm -rf ${1}.Ma
    fi
    if
        [ -f ${1}.Sy ]
    then
        grep -v "[\*#]ident" ${1}.Sy |
                    sed "${SEDCMD1}" > System
        rm -rf ${1}.Sy
    fi
    if
        [ -f ${1}.Mf ]
    then
        grep -v "[\*#]ident" ${1}.Mf > Master
        rm -rf ${1}.Mf
    fi
    if
        [ -f ${1}.Sf ]
    then
        grep -v "[\*#]ident" ${1}.Sf |
                    sed "${SEDCMD1}" > Sfsys
        rm -rf ${1}.Sf
    fi
    if
        [ -f ${1}.No ]
    then
        grep -v "[\*#]ident" ${1}.No > Node
        rm -rf ${1}.No
    fi
    if
        [ -f ${1}.Rc ]
    then
        grep -v "[\*#]ident" ${1}.Rc > Rc
        rm -rf ${1}.Rc
    fi
    echo "Installing ${NAME} ${1} module ..."
    ${CONFBIN}/idcheck -p ${1} > ${ERR} 2>&1
RET=$?
    if [ ${RET} = 0 ] || [ ${RET} = 8
    then
        ${CONFBIN}/idinstall -a ${1} > ${ERR} 2>&1
        RET=$?
    else
        ${CONFBIN}/idinstall -u ${1} > ${ERR} 2>&1
        RET=$?
```

```
□□□□fi
□□□□rm□-rf□${IDCOMPS}
□□□□if□[□${RET}□!=□0□]
□□□□then
□□□□□□□□□□□□□message□"The□installation□cannot□be□completed□due□to□\
an□error□in□the□driver□installation□during□the□installation□of□the□${1}□\
module□of□the□${NAME}.□The□file□${ERR}□contains□the□error."
□□□□□□□□□□□□□□□□□□□□□□□□exit□${FAILURE}
□□□□fi
}
FAILURE=1□□□#□fatal□error
CONFDIR=/etc/conf
CONFBIN=${CONFDIR}/bin
NOTOOLS="ERROR:□The□installable□driver□feature□has□been□removed.□\
The□${NAME}□cannot□be□installed."
SEDCMD1="s/[□□□□□□□□□□]N[□□□□□]/□□□□Y□□□□□□□□□/"
SEDCMD2="s/[□□□□□□□□□□]N/□□□□Y/"
#□determine□that□the□ID/TP□tools□are□available
if
□□□□[□!□-x□${CONFBIN}/idbuild□-o□!□-x□${CONFBIN}/idinstall□\
□□□□□□□□□□□□□-o□!□-x□${CONFBIN}/idcheck□-o□!□-x□${CONFBIN}/idinstallf□]
then
□□□□message□${NOTOOLS}
□□□□exit□$FAILURE
fi
VOLATILES=`echo□${WHERE}/drvdemo/*`
cd□${WHERE}/drvdemo
#□create□driver□node□file
#□node□are□create□by□idmknod
node=0
while□[□$node□-le□$NDEVICE□]
do
echo□"xxx□xxx/xxx${node}□c□${node}"
node=□`expr□$node□+□1`
done□>$WHERE/drvdemo/xxx.No
#□install□drivers/modules
for□MODULE□in□xxx
do
□□□□do_install□${MODULE}
□□□□□□□□${CONFBIN}/idinstall□${MODULE}
□□□□□□□□${CONFBIN}/idinstallf□-f□${PKGINST}□${MODULE}
done
${CONFBIN}/idbuild
removef□${PKGINST}□${VOLATILES}□>/dev/null□2>&1
removef□${PKGINST}□${WHERE}/drvdemo>dev/null□2>&1
removef□-f□${PKGINST}□>/dev/null□2>&1
cd□/
rm□-rf□${WHERE}/drvdemo
exit□10
```

4.4.3 Case #3: Creating a database file at installation time

This study creates a database file at the time of installation and saves a copy of the database when the package is removed.

Techniques

This case study shows examples of the following techniques:

- using classes and class action scripts to perform special actions on different sets of objects
- using the *space* file to inform *pkgadd* that extra space will be required to install this package properly
- using the *installf* command

Approach

To create a database file at the time of installation and save a copy on removal, you must:

1. Create three classes.

   This package requires three classes:

   - the standard class of *none* (contains a set of processes belonging in the subdirectory *bin*)
   - the *admin* class (contains an executable file *config* and a directory containing data files)
   - the *cfgdata* class (contains a directory)

2. Make the package collectively relocatable.

   Notice in the *prototype* file (Figure "Case #3 prototype file" on ...) that none of the pathnames begin with a slash or a variable. This indicates that they are collectively relocatable.

3. Calculate the amount of space the database file will require and create a *space* file to deliver with the package. This file notifies *pkgadd* that this package requires extra space and how much extra space. Figure "Case #3 space file" on ...shows the *space* file for this package.

4. Create an installation class action script for the *admin* class.

   The script, shown in the figure "Case #3 installation class action script (i.admin)" on ..., initializes a database using the data files belonging to the *admin* class. To perform this task, it:

   - copies the source data file to its proper destination
   - creates an empty file named *config.data* and assigns it to a class of *cfgdata*
   - executes the *bin/config* command (delivered with the package and already installed) to populate the database file *config.data* using the data files belonging to the *admin* class
   - executes *installf –f* to finalize installation

   No special action is required for the *admin* class at removal time so no removal class action script is created. This means that all files and directories in the *admin* class will simply be removed from the system.

5. Create a removal class action script for the *cfgdata* class.

   The script, shown in the figure "Case #3 removal class action script (r.cfgdata)" on ..., makes a copy of the database file before it is deleted during package removal. No special action is required for this class at installation time, so no installation class action script is needed.

   Remember that the input to a removal script is a list of pathnames to remove. Pathnames always appear in lexical order with the directories appearing first. This script captures directory names so that they can be acted upon later and copies any files to a directory named */tmp*. When all of the pathnames have been processed, the script then goes back and removes all directories and files associated with the *cfgdata* class.

   The outcome of this removal script is to copy *config.data* to */tmp* and then remove the *config.data* file and the data directory.

Sample files

Case #3 pkginfo file:

```
PKG='krazy'
NAME='KrAzY□Applications'
CATEGORY='applications'
ARCH='i386'
VERSION='Version□1'
CLASSES='none□cfgdata□admin'
```

Case #3 prototype file:

```
i□pkginfo
i□request
i□i.admin
i□r.cfgdata
d□none□bin□555□root□sys
```

```
f□none□bin/process1□555□root□other
f□none□bin/process2□555□root□other
f□none□bin/process3□555□root□other
f□admin□bin/config□500□root□sys
d□admin□cfg□555□root□sys
f□admin□cfg/datafile1□444□root□sys
f□admin□cfg/datafile2□444□root□sys
f□admin□cfg/datafile3□444□root□sys
f□admin□cfg/datafile4□444□root□sys
d□cfgdata□data□555□root□sys
```

Case #3 space file:

```
#□extra□space□required□by□config□data□which□is
#□dynamically□loaded□onto□the□system
data□500□1
```

Case #3 installation class action script (i.admin):

```
#□PKGINST□parameter□provided□by□installation□service
#□BASEDIR□parameter□provided□by□installation□service
while□read□src□dest
do
□□□□□□□□□□#□the□installation□service□provides□'/dev/null'□as□the
□□□□□□□□□□#□pathname□for□directories,□pipes,□special□devices,□etc
□□□□□□□□□□#□which□it□knows□how□to□create
□□□□□□□□□□[□"$src"□=□/dev/null□]□&&□continue
□□□□□□□□□cp□$src□$dest□||□exit□2
done
#□if□this□is□the□last□time□this□script□will
#□be□executed□during□the□installation,□do□additional
#□processing□here
if□[□"$1"□=□ENDOFCLASS□]
then
□□□□□□□□□□#□our□config□process□will□create□a□data□file□based□on□any□changes
□□□□□□□□□□#□made□by□installing□files□in□this□class;□make□sure
□□□□□□□□□□#□the□data□file□is□in□class□'cfgdata'□so□special□rules□can□apply
□□□□□□□□□□#□to□it□during□package□removal
□□□□□□□□□□installf□-c□cfgdata□$PKGINST□$BASEDIR/data/config.data□f□444□root□sys□||
□□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□□□$BASEDIR/bin/config□>□$BASEDIR/data/config.data□||
□□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□□□installf□-f□-c□cfgdata□$PKGINST□||
□□□□□□□□□□□□□□□□□□□exit□2
fi
exit□0
```

Case #3 removal class action script (r.cfgdata):

```
#□the□product□manager□for□this□package□has□suggested□that
#□the□configuration□data□is□so□valuable□that□it□should□be
#□backed□up□to□/tmp□before□it□is□removed!
while□read□path
do
□□□□□□□□□□#□pathnames□appear□in□lexical□order,□thus□directories
□□□□□□□□□□#□will□appear□first;□you□can't□operate□on□directories
□□□□□□□□□□#□until□done,□so□just□keep□track□of□names□until
□□□□□□□□□□#□later
□□□□□□□□□□if□[□-d□$path□]
□□□□□□□□□□then
□□□□□□□□□□□□□□□□□□□dirlist="$dirlist□$path"
□□□□□□□□□□□□□□□□□□□continue
```

```
□□□□□□□□□fi
□□□□□□□□□mv□$path□/tmp□||□exit□2
done
if□[□-n□"$dirlist"□]
then
□□□□□□□□□rm□-rf□$dirlist□||□exit□2
fi
exit□0
```

4.4.4 Case #4: Using optional packaging files

This package uses the optional packaging files to define package compatibilities and dependencies and to present a copyright message during installation.

Techniques

This case study shows examples of the following techniques:

- using the *copyright* file
- using the *compver* file
- using the *depend* file

Approach

To meet the requirements in the description, you must:

1. Create a *copyright* file.

   A *copyright* file contains the ASCII text of a copyright message. The message shown in the figure "Case #4 copyright file" on ... will be displayed on the screen during package installation (and also during package removal).

2. Create a *compver* file.

   The *pkginfo* file shown in the figure "Case#4 pkginfo file" on ... defines this package version as version 3.0. The *compver* file, shown in the figure "Case #4 compver file" on ..., defines version 3.0 as being compatible with versions 2.3, 2.2, 2.1, 2.1.1, 2.1.3 and 1.7.

3. Create a *depend* file.

   Files listed in a *depend* file must already be installed on the system when a package is installed. The example shown in the figure "Case #4 depend file" on ... has 11 packages which must already be on the system at installation time.

Sample files

Case #4 pkginfo file:

```
PKG='case4'
NAME='Case□Study□#4'
CATEGORY='application'
ARCH='i386'
VERSION='Version□3.0'
CLASSES='none'
```

Case #4 copyright file:

```
Copyright (c) Siemens Nixdorf Informationssysteme AG 1993
All Rights Reserved.
```

Case #4 compver file:

```
Version 2.3
Version 2.2
Version 2.1
Version 2.1.1
```

Version 2.1.3
Version 1.7

Case #4 depend File:

P□acu□□□□Advanced□C□Utilities
□□□□□□□□□Issue□4□Version□1
P□cc□□□□□C□Programming□Language
□□□□□□□□□Issue□4□Version□1
P□dfm□□□□Directory□and□File□Management□Utilities
P□ed□□□□□Editing□Utilities
P□esg□□□□Extended□Software□Generation□Utilities
□□□□□□□□□Issue□4□Version□1
P□graph□□Graphics□Utilities
P□nfs□□□□Remote□File□Sharing□Utilities
□□□□□□□□□Issue□1□Version□1
P□rx□□□□□Remote□Execution□Utilities
P□sgs□□□□Software□Generation□Utilities
□□□□□□□□□Issue□4□Version□1
P□shell□□Shell□Programming□Utilities
P□sys□□□□System□Header□Files
□□□□□□□□□Release□3.1

4.4.5 Case #5a: Modifying existing data files - using the sed class and a postinstall script

This study modifies a file which exists on the installation machine during package installation. It uses one of three modification methods. The other two methods are shown in Cases 5b and 5c. The file modified is */etc/inittab*. As a rule you should not modify */etc/inittab*. In this example we simply use it to demonstrate the pros and cons of the actions in the various class scripts. */etc/inittab* is rebuilt from the files in */etc/conf/init.d* and */etc/conf/cf.d/init.base* after each kernel rebuild at boot time. So *inittab* entries should be implemented only in the corresponding files in */etc/conf/init.d*.

Techniques

This case study shows examples of the following techniques:

- using the *sed* class
- using a *postinstall* script

Approach

To modify */etc/inittab* at the time of installation, you must:

1. Add the *sed* class script to the *prototype* file.

   The name of a script must be the name of the file that will be edited. In this case, the file to be edited is */etc/inittab* and so our *sed* script is named */etc/inittab*. There are no requirements for the *mode owner group* of a *sed* script (represented in the sample *prototype* by question marks). The file type of the *sed* script must be *e* (indicating that it is editable). The *prototype* file for this case study is shown in the figure "Case #5a pkginfo file" on ....

2. Set the *CLASSES* parameter to include *sed*.

   In the case of the example shown in the figure "Case #5a prototype file" on ..., *sed* is the only class being installed. However, it could be one of any number of classes.

3. Create a *sed* class action script.

   You cannot deliver a copy of */etc/inittab* that looks the way you need for it to, since */etc/inittab* is a dynamic file and you have no way of knowing how it will look at the time of package installation. Using a *sed* script allows us to modify the */etc/inittab* file during package installation.

   As already mentioned, the name of a *sed* script should be the same as the name of the file it will edit. A *sed* script contains *sed* commands to remove and add information to the file. See the figure "Case #5a sed

script (/etc/inittab)" on ... for an example *sed* script.

4. Create a *postinstall* script.

   You need to inform the system that */etc/inittab* has been modified by executing *init q*. The only place you can perform that action in this example is in a *postinstall* script. Looking at the example *postinstall* script, you will see that its only purpose is to execute *init q*.

This approach to editing */etc/inittab* during installation has two drawbacks. First of all, you have to deliver a full script (the *postinstall* script) simply to perform *init q*. In addition to that, the package name at the end of each comment line is hardcoded. It would be nice if this value could be based on the package instance so that you could distinguish between the entries you add for each package.

Sample files

Case #5a pkginfo file:

```
PKG='case5a'
NAME='Case Study #5a'
CATEGORY='applications'
ARCH='i386'
VERSION='Version 1d05'
CLASSES='sed'
```

Case #5a prototype file:

```
i pkginfo
i postinstall
e sed /etc/inittab ? ? ?
```

Case #5a sed script (/etc/inittab):

```
!remove
# remove all entries from the table that are associated
# with this package, though not necessarily just
# with this package instance
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d
!install
# remove any previous entry added to the table
# for this particular change /^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d
# add the needed entry at the end of the table;
# sed(1) does not properly interpret the '$a'
# construct if you previously deleted the last
# line, so the command
#        $a\
#        rb:023456:wait:/usr/robot/bin/setup #ROBOT
# will not work here if the file already contained
# the modification.  Instead, you will settle for
# inserting the entry before the last line!
$i\
rb:023456:wait:/usr/robot/bin/setup #ROBOT
```

Case #5a postinstall script:

```
# make init re-read inittab
/sbin/init q ||
        exit 2
exit 0
```

4.4.6 Case #5b: Modifying existing data files - using class action scripts and creating classes

This study modifies a file which exists on the installation during package installation. It uses one of three

modification methods. The other two methods are shown in Cases 5a and 5c. The file modified is */etc/inittab*. As a rule you should not modify */etc/inittab*. In this example we simply use it to demonstrate the pros and cons of the actions in the various class scripts.

Techniques

This case study shows examples of the following techniques:

- creating classes
- using installation and removal class action scripts

Approach

To modify */etc/inittab* during installation, you must:

1. Create a class.

   Create a class called *inittab*. You must provide an installation and a removal class action script for this class. Define the *inittab* class in the *CLASSES* parameter in the *pkginfo* file (as shown in the figure "Case #5b pkginfo file" on ...).

2. Create an *inittab* file.

   This file contains the information for the entry that you will add to */etc/inittab*. Notice in the *prototype* file (Figure "Case #5b prototype file" on ...) that *inittab* is a member of the *inittab* class and has a file type of *e* for editable. Figure "Case #5b installation class action script (i.inittab)" on ... shows what *inittab* looks like.

3. Create an installation class action script.

   Since class action scripts must be multiply executable (meaning you get the same results each time they are executed), you can't just add our text to the end of the file. The script, shown in the figure "Case #5b installation class action script (i.inittab)" on ..., performs the following procedures:

   - checks to see if this entry has been added before
   - if it has, removes any previous versions of the entry
   - edits the *inittab* file and adds the comment lines so you know where the entry is from
   - moves the temporary file back into */etc/inittab*
   - executes *init q* when it receives the end-of-class indicator

   Note that *init* q can be performed by this installation script. A one-line postinstall script is not needed by this approach.

4. Create a removal class action script.

   The removal script, shown in the figure "Case #5b removal class action script (r.inittab)" on ..., is very similar to the installation script. The information added by the installation script is removed and *init q* is executed.

This case study resolves the drawbacks to Case 5a. You can support multiple package instances since the comment at the end of the *inittab* entry is now based on package instance. Also, you no longer need a one-line postinstall script. However, this case has a drawback of its own. You must deliver two class action scripts and the *inittab* file to add one line to a file. Case 5c shows a more streamlined approach to editing */etc/inittab* during installation.

Note that files of type *e* require the presence of an installation and a removal script. If there is no removal script, the original will be removed with potentially fatal consequences.

Sample files

Case #5b pkginfo file:

```
PKG='case5b'
NAME='Case Study #5b'
CATEGORY='applications'
ARCH='i386'
```

```
VERSION='Version 1d05'
CLASSES='inittab'
```

Case #5b prototype file:

```
i pkginfo
i i.inittab
i r.inittab
e inittab /etc/inittab ? ?
```

Case #5b installation class action script (i.inittab):

```
# PKGINST parameter provided by installation service
while read src dest
do
 sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" $dest > /tmp/$$itab ||
                exit 2
          # remove all entries from the table that
          # associated with this PKGINST
 sed -e "s/$/#$PKGINST" $src >> /tmp/$$itab ||
                exit 2
          mv /tmp/$$itab $dest ||
                exit 2
done
if [ "$1" = ENDOFCLASS ]
then
          /sbin/init q ||
                exit 2
fi
exit 0
```

Case #5b removal class action script (r.inittab):

```
# PKGINST parameter provided by installation service
while read src dest
do
          # remove all entries from the table that
          # are associated with this PKGINST
          sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" $dest > /tmp/$$itab ||
                exit 2
mv /tmp/$$itab $dest ||
                exit 2
done
/sbin/init q ||
          exit 2
exit 0
```

Case #5b inittab File:

```
rb:023456:wait:/usr/robot/bin/setup
```

4.4.7 Case #5c: Modifying existing data files - using the build class

This study modifies a file which exists on the installation machine during package installation. It uses one of three modification methods. The other two methods are shown in Cases 5a and 5b. The file modified is */etc/inittab*. As a rule you should not modify */etc/inittab*. In this example we simply use it to demonstrate the pros and cons of the actions in the various class scripts.

Techniques

This case study shows examples of the following technique:

- using the *build* class

Approach

This approach to modifying */etc/inittab* uses the *build* class. A *build* class file is executed as a shell script and its output becomes the new version of the file being executed. In other words, the data file *inittab* that is delivered with this package will be executed and the output of that execution will become */sbin/inittab*.

The *build* class file is executed during package installation and package removal. The argument *install* is passed to the file if it is being executed at installation time. Notice in the sample *build* file in the figure "Case #5c build script (/sbin/init)" on ... that installation actions are defined by testing for this argument.

To edit */etc/inittab* using the *build* class, you must:

1. Define the *build* file in the *prototype* file.

   The entry for the *build* file in the *prototype* file should place it in the *build* class and define its file type as *e*.

   Note that files of type *e* require the presence of an installation and a removal script. If there is no removal script, the original will be removed with potentially fatal consequences.

   Be certain that the *CLASSES* parameter in the *pkginfo* file is defined as *build*. Figure "Case #5c pkginfo file" on ... shows the *pkginfo* file for this example and the figure "Case #5c prototype file" on ... shows the *prototype* file.
2. Create the *build* file.

   The *build* file shown in the figure "Case #5c build script (/sbin/init)" on ... performs the following procedures:

   - Edits */etc/inittab* to remove any changes already existing for this package. Notice that the filename */etc/inittab* is hardcoded into the *sed* command.
   - If the package is being installed, adds the new line to the end of */etc/inittab*. A comment tag is included in this new entry to remind us from where that entry came.
   - Executes *init q*.

This solution addresses the drawbacks in case studies 5a and 5b. Only one file is needed (beyond the *pkginfo* and *prototype* files), that file is short and simple, it works with multiple instances of a package since the *$PKGINST* parameter is used, and no *postinstall* script is required since *init q* can be executed from the *build* file.

Sample files

Case #5c pkginfo file:

```
PKG='case5c'
NAME='Case Study #5c'
CATEGORY='applications'
ARCH='i386'
VERSION='Version 1d05'
CLASSES='build'
```

Case #5c prototype file:

```
i pkginfo
e build /etc/inittab ? ? ?
```

Case #5c build script (/sbin/init):

```
#□PKGINST□parameter□provided□by□installation□service
#□remove□all□entries□from□the□existing□table□that
#□are□associated□with□this□PKGINST
sed□-e□"/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d"□/etc/inittab□||
□□□□□□□□exit□2
```

```
if [ "$1" = install ]
then
        #add the following entry to the table
        echo "rb:023456:wait:/usr/robot/bin/setup #$PKGINST" ||
                exit 2
fi
/sbin/init q ||
        exit 2
exit 0
```

4.4.8 Case #6: Using classes and class action scripts

This case study modifies a number *crontab* files during package installation.

Techniques

This case study shows examples of the following techniques:

- using classes and class action scripts
- using the *crontab* command within a class action script

Approach

You could use the *build* class and follow the approach shown for editing */etc/inittab* in case study 5c except that you want to edit more than one file. If you used the *build* class approach, you would need to deliver one for each *cron* file edited. Defining a *cron* class provides a more general approach. To edit a *crontab* file with this approach, you must:

1. Define the *cron* files that will be edited in the *prototype* file.

   Create an entry in the *prototype* file for each *crontab* file which will be edited. Define their class as *cron* and their file type as *e*. Use the actual name of the file to be edited, as shown in the figure "Case #6 prototype file" on ....

2. Create the *crontab* files that will be delivered with the package.

   These files contain the information you want added to the existing *crontab* files of the same name. See the figures "Case #6 root crontab file" on ... and "Case #6 sys crontab file" on ... for examples of what these files look like.

3. Create an installation class action script for the *cron* class.

   The *i.cron* script (Figure "Case #6 installation class action script (i.cron)" on ...) performs the following procedures:

   - Calculates the user id.

     This is done by setting the variable *user* to the base name of the *cron* class file being processed. That name equates to the user id. For example, the basename of */var/spool/cron/crontabs/root* is *root* (which is also the user id).

   - Executes *crontab* using the user id and the *-l* option.

     Using the *-l* option tells *crontab* to send the standard output the contents of the *crontab* for the defined user.

   - Pipes the output of the *crontab* command to a *sed* script that removes any previous entries that have been added using this installation technique.
   - Puts the edited output into a temporary file.
   - Adds the data file for the *root* user id (that was delivered with the package) to the temporary file and adds a tag so that you will know from where these entries came.
   - Executes *crontab* with the same user id and gives it the temporary file as input.

4. Create a removal class action script for the *cron* class.

   The removal script, shown in the figure "Case #6 removal class action script (r.cron)" on ..., is the same as

the installation script except that there is no procedure to add information to the *crontab* file.

These procedures are performed for every file in the *cron* class.

Sample files

Case #3 pkginfo file:

```
PKG='case6'
NAME='Case Study #6'
CATEGORY='application'
ARCH='i386'
VERSION='Version 1.0'
CLASSES='cron'
```

Case #6 prototype file:

```
i pkginfo
i i.cron
i r.cron
e cron /var/spool/cron/crontabs/root ? ? ?
e cron /var/spool/cron/crontabs/sys ? ? ?
```

Case #6 installation class action script (i.cron):

```
#□PKGINST□parameter□provided□by□installation□service
while□read□src□dest
do
□□□□□□□□user=`basename□$dest`□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□(crontab□-l□$user□|
□□□□□□□□sed□-e□"/#$PKGINST$/d"□>□/tmp/$$crontab)□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□sed□-e□"s/$/#$PKGINST/"□$src□>>□/tmp/$$crontab□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□su□$user□-c□crontab□<□/tmp/$$crontab□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□rm□-f□/tmp/$$crontab
done
exit□0
```

Case #6 removal class action script (r.cron)

```
#□PKGINST□parameter□provided□by□installation□service
while□read□path
do
□□□□□□□□user=`basename□$path`□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□(crontab□-l□$user□|
□□□□□□□□sed□-e□"/#$PKGINST$/d"□>□/tmp/$$crontab)□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□su□$user□-c□crontab□<□/tmp/$$crontab□||
□□□□□□□□□□□□□□□□□□exit□2
□□□□□□□□rm□-f□/tmp/$$crontab
done
exit□0
```

Case #6 root crontab file (delivered with package):

```
41,1,21□*□*□*□*□/usr/lib/uucp/uudemon.hour□>□/dev/null
45□23□*□*□*□ulimit□5000;□/usr/bin/su□uucp□-c□"/usr/lib/uucp/uudemon.cleanup"□>
```

```
/dev/null□2>&1
11,31,51□*□*□*□*□/usr/lib/uucp/uudemon.poll□>□/dev/null
```

Case #6 sys crontab file (delivered with package):

```
0□*□*□*□0-6□/usr/lib/sa/sa1
20,40□8-17□*□*□1-5□/usr/lib/sa/sa1
5□18□*□*□1-5□/usr/lib/sa/sa2□-s□8:00□-e□18:01□-i□1200□-A
```

5 Device drivers

This section defines procedures for writing and packaging a device driver for Reliant UNIX 5.43. It contains general information on "generic" Reliant UNIX System device drivers. Also described is the Installable Driver (ID) scheme for Reliant UNIX 5.43. ID allows users to add peripheral devices via a medium (such as a floppy disk, cartridge tape or CD-ROM) containing a Driver Software Package (DSP). Users will install and remove DSPs by using the *pkgadd* and *pkgrm* commands. This section also provides the implementation-dependent information for Reliant UNIX 5.43. Additional generic driver reference material can be found in [16].

It is assumed that the reader has user-level experience with the Reliant UNIX System, some general knowledge of Reliant UNIX System concepts, and the ability to write sophisticated C language programs. Writing a device driver carries a heavy responsibility. As part of the Reliant UNIX Operating System kernel, a device driver is assumed to always take the correct action. Few limits are placed on the driver by the other parts of the kernel, and the driver must be written to never compromise the system's stability.

5.1 What is a Reliant UNIX device driver?

The Reliant UNIX Operating System kernel can be divided into two parts: the first part deals with management of the file system and processes, and the second part deals with the management of devices, such as terminals, disks, tape drives, and network media. To simplify the terminology, this section will refer to the first part as the kernel, although strictly speaking, drivers are part of the kernel too. The discussion here will focus on the second part that contains the drivers, sometimes called the I/O subsystem.

Associated with each device is a piece of code, called the device driver, that manages the device hardware. The device driver is responsible for bringing the device into and out of service, setting hardware parameters in the device, transmitting data from the kernel to the device, receiving data from the device and passing it back to the kernel, and handling device errors.

One strength of the Reliant UNIX System is the ease with which new drivers can be integrated with existing software. The integration process is simple because the operating system architecture provides a uniform software interface to drivers. Processes use the same model when communicating with disks, terminals, printers or even "pseudo" devices that exist only in software. Every device on a Reliant UNIX System looks like a file. In fact, the user-level interface to the device is called a " special file."

The device special files reside in the */dev* directory, and a simple *ls* will tell you quite a bit about the device. For example, the command *ls -l /dev/lp* will yield the following on Reliant UNIX 5.43:

crw-rw-rw□□1□root□□root□□□□7,□□1□□Nov□26□12:33□□lp

This says that the "lp" (line printer) is a character type device (the first letter of the file mode field is "c"), and that major number 7, minor device 1 is assigned to the device. More will be said about device types, both major and minor numbers, later.

5.2 The generic Reliant UNIX driver

This section addresses issues relevant to drivers on any Reliant UNIX System. Throughout this section, references are made to how things work on a "generic" or traditional Reliant UNIX System, along with some specific details on how Reliant UNIX 5.43 is implemented. The areas of device interrupts and priority levels in particular are heavily machine-dependent and reflect Reliant UNIX 5.43 implementation.

Reliant UNIX System device drivers for different computer systems have many identical characteristics. However, even on the same machine, one driver may be very different from another because of the wide spectrum of functions that drivers perform. Let's first discuss some design issues and examine the common features.

Not all portions of this section will be appropriate for STREAMS drivers and modules. STREAMS programmers are encouraged to use the "STREAMS Programmer's Guide" [17] as a principal reference and use only those parts of this section that pertain to machine specifics and driver installation.

5.3 Driver activities and responsibilities

A user process runs in a space isolated from critical system data and other programs, protecting the system and other programs from its mistakes. In contrast, a driver executes in kernel mode, placing few limits on its freedom of action. The driver is simply assumed to be correct and responsible.

This level of responsibility and reliability cannot be avoided. A driver must be part of the kernel to service interrupts and access device hardware. The existence of the driver is one of the major factors that permits the kernel to present a uniform interface for all devices and to protect processes from some kinds of errors.

The importance of reliable driver code is clear. The driver must not make mistakes that hurt any portion of the system. It should process interrupts efficiently to preserve the scheduler's ability to balance demands on the system. It should use system buffers responsibly to avoid degrading system performance or requiring that more space be devoted to buffers than is really needed.

This section provides a broad overview of what device drivers do inside the Reliant UNIX Operating System kernel. The specific details are provided later. The purpose of the overview is to introduce issues of significance and establish a common language for further discussion. Experienced driver developers will be familiar with much of the information, but those new to Reliant UNIX System device drivers may find the implications of a multi-tasking environment more complex than expected.

5.3.1 System buffers

A feature common to most drivers is their use of buffers. There are three types of buffers in a standard Reliant UNIX 5.43: system buffers, STREAMS messages, and Kernel Memory Allocator (KMA) buffers. They differ greatly in size and structure and are meant to fulfill different needs.

System buffers are the size of a file system block. The size of the file system block is dependent on the type of the file system. It can vary from 512 bytes to 8K. This buffer pool primarily supports disk I/O operations. STREAMS messages are for use by drivers written to the STREAMS interface. They are allocated for the driver through the kernel utilities, so the driver does not need to allocate a pool of its own messages. Kernel Memory Allocator(KMA) buffers are "borrowed" by the driver from a common memory pool that is used by all parts of the kernel. All types of drivers may use them.

All of these types buffers are a commonly used Reliant UNIX System resources. Every driver should be written with the finite nature of the machine in mind; intense buffer use by a driver can reduce the performance of other drivers or require more memory be devoted to buffers. When more memory or space is allocated to buffers, the memory or space available for user processes is correspondingly decreased. More will be said later about how to obtain and return buffers.

5.3.2 Data transfer between system and user space

The kernel instruction and data spaces are strictly segregated from those of user processes. The need for the kernel to protect itself is obvious. This protection creates the need for a way to transfer information from user space to kernel space and back.

There are several routines for transferring data across the user/system boundary. Some transfer bytes, some transfer words, and others transfer arbitrary size buffers. Each type of operation implies a pair of routines: one for transfers from user space to system space and one for those in the opposite direction.

At this time, it would be helpful to consider a representative I/O operation and the information transfer across the user/kernel boundary it generates. As an example, take a request from a process to write a buffer on the disk. The *write* routine takes the file descriptor, the buffer address in user space, and the length of the data in the buffer as parameters.

The system call causes the processor to transfer from user to kernel mode, and to execute the *write* routine in the generic file interface. When *write()* realizes that the file is "special" (a device), it uses the appropriate switch table (defined later in Section "Major and minor numbers") to select the corresponding routine associated with the device. The device driver's *write* routine is then faced with a decision.

Since the disk is a shared resource, the device driver may not find it convenient or possible to do the requested *write* just when it is requested. However, when the system call returns, the process assumes that the operation is complete and may do whatever it wishes with its buffer. If the kernel wishes to defer the *write* to disk, it must take a copy of the information from user space, keeping it in system space until the *write* can

be done.

5.3.3 Sleeping and waking processes

In the previous section, an example of a *write* operation to the disk introduced several basic concepts. A process might have to wait for the requested information to be read or written from/to the disk before continuing. One way that processes can coordinate their actions with events is through the *sleep()* and *wakeup()* calls.

Let's consider a read operation in greater detail. When the request is made, the driver has some calculations and setup functions to perform. After these are complete, the request for the information can be made, but there will be a delay before the information is available. The delay will, at a minimum, be due to the retrieval time for the disk. However, it could be much longer than that if other requests are queued ahead of this one.

Since Reliant UNIX 5.43 is a multi-user, multi-tasking operating system, it is possible that another job is ready to run and waiting for a chance to use the machine. One waiting process should not occupy the machine while another process is ready to run, so some way must be found to have the waiting process release the machine for the executable process until the information the waiting process needs to be able to continue is available. The Sleep/Wakeup mechanism can coordinate this. In the disk access example, the read routine in the disk's driver set would issue a request for the information and put the process to "sleep."

A sleeping process is still considered to be an active process but is kept on a queue of jobs whose execution is suspended while they wait for a particular event. When the process goes to sleep, it specifies the event that must occur before it may continue its task. This event is represented by an identifier, typically an address of a structure associated with the transaction. The *sleep()* call records the process number and the event, then places it on the list of sleeping processes. Control of the machine is then transferred to the highest priority runnable process.

When the data transfer completes, the disk will post an interrupt, causing the interrupt routine in the driver to be activated. The interrupt routine will do whatever is required to properly service the device and issue a *wakeup()* call. It must know what identifier was used by the process as the sleeping event to wake it. This scenario for coordination between asynchronous events appears throughout the kernel.

5.3.4 Kernel timers

In some cases, a driver must be sure that it is awakened after a maximum period. For those situations where a limit must be placed on how long a process will sleep, the *timeout()* facility is available.

This routine takes three arguments: an integer function pointer, a character pointer, and an integer. The integer specifies the period of time in "ticks". The defined constant *HZ* gives the interrupt frequency used by a given kernel, which means that *HZ* is the number of ticks corresponding to one second. When this period of time has passed, the function pointed to by the first argument to *timeout()* will be called with the second argument as its parameter.

A driver can ensure that it will be able to resume its execution (even if no call to *wakeup()* is made) by first calling *timeout()* and then *sleep()*. This should be done, however, only if truly necessary, as it carries some heavy processing requirements. When the call to *timeout()* is made, it inserts the specified event into the callout table.

If the sleeping process is not awakened before the "timeout" event, the specified function will be called. The second argument to the *timeout()* routine could be the event the driver was about to sleep on. When the function is called, it can use this information to call *wakeup()* to wake the driver. The function called from the callout table should also set some internal flag to permit the driver to distinguish between the two ways it can be awakened. When the expected event occurs, the entry shoul be removed from the callout table by *untimeout()*.

5.3.5 Synchronous and interrupt sections of a driver

As described earlier, the system uses system buffers and routines to transfer information across the user/system boundary.

The interrupt portion of the driver is driven by interrupts from devices. The rest of the driver executes only

when the process talking to the driver is the active process. The execution of this part of the driver is synchronized with the process it serves and will be called the synchronous portion of the driver.

Since the synchronous portion of the driver has the proper process context, it is responsible for organizing the information required for the requested operation. It is responsible for any transfer of information across the user/system boundary. When the request has been properly submitted, the synchronous portion of the driver can do nothing but wait until the requested operation is complete, so it sleeps.

The interrupt driven section of the driver responds to the demands of the device as they come. The synchronous part must leave enough information in common data structures to permit the interrupt routine to figure out what is happening. The interrupt routine is called when an operation is complete. It is responsible for servicing the device and waking the process waiting on the event. Note that the interrupt routine can be called at any time. It cannot engage in any activity that depends on process context.

5.3.6 Interrupt processing

The previous section defined the interrupt and synchronous portions of a driver and mentioned that the interrupt portion is driven by interrupts from devices, which require attention from the driver.

When a device requests some software service, it generates an "interrupt." Each device can interrupt the system at a specific "priority level." If the currently executing code has not blocked interrupts at that level, it will immediately save its status and "trap" to an interrupt handler. The interrupt routine in the driver must determine the cause of the interrupt and take appropriate action. If the synchronous portion of the driver was waiting for this event, the interrupt routine should issue a call to *wakeup()*.

5.3.7 Critical sections of the driver

The discussion so far has been centered around interrupts occurring in isolation. Interrupts from all devices on the system can occur at any time, and the complex implications of this are important. The relationship between the synchronous and interrupt portions of the driver are affected, as are those between drivers sharing data.

When two sections of kernel code have a common interest in specific data, they must be careful to coordinate their efforts. If an interrupt switches control of the system to the interrupt driven portion of the driver, then manipulation of the common data might be caught in the midst of its work. This would render the information invalid and inconsistent.

These concerns are grouped under the general heading *critical sections*. The importance of the issue is clear; the integrity and accuracy of the data used by drivers is at stake. The word *sections* refers to the portions of code that manipulate the common data, rather than the data itself. Thus, a *critical section* of code is one that manipulates data that is of concern to another piece of code capable of interrupting the first.

A routine in the kernel that has a critical section must have a way to protect itself from being interrupted when manipulating critical data. A set of subroutines that permit code to Set the Priority Level (*spl*) of the processor solve the problem and are listed in Section "Setting processor priority levels". A clear understanding of the need for these routines can be achieved only by examining a detailed scenario.

Imagine a section of code in the synchronous portion of a driver that manipulates status flags. Such flags are frequently used to communicate between the synchronous and interrupt portions of a driver. Consider also that the interrupt portion has code that manipulates those flags.

Consider what happens if the synchronous portion of the driver receives a request that requires it to manipulate the values of several flags, but in the midst of the manipulation, the device gives an interrupt, transferring control to the interrupt portion of the driver. The interrupt routine decides that it must consult the flag values to make some decision and then set them to new values.

The flags are in the incorrect state because the synchronous routine has only half finished changing them when the interrupt routine took over. This may cause the interrupt routine to go mad, or it may simply make an innocuous but incorrect decision. Assume that the interrupt routine does not run amok but simply looks at the flags, makes decisions, and changes a couple of flag values. Then, when the interrupt returns, the synchronous portion of the code, unaware that it was interrupted, finishes the changes it had started.

Whether the data manipulated in a critical section is changed by the interrupting routine is unimportant. Any

portion of code that can be interrupted and that manipulates data of interest to the interrupting code is a *critical section*. When a critical section is identified, it can be protected from interruption by a call to an *spl* routine of the appropriate level.

5.3.8 How data moves between the kernel and the device

The discussions above did not examine how the data moves between the memory accessible to the kernel and the device itself. This is a machine-dependent detail, but it is instructive to examine how this is done. Some machines require the central processing unit (CPU) to execute special I/O instructions to move data between a device register and addressable memory. If a device moves data to or from memory without the aid of the CPU, this process is known as direct memory access (DMA). Another scheme, known as memory mapped I/O, implements the device interface as one or more locations in the memory address space. All of these schemes are used on Reliant UNIX 5.43, but the most common method uses I/O instructions.

The operating system usually provides function calls that let drivers access the data in a general way. Reliant UNIX 5.43 for the 80x86 Architecture implementation provides *inb()* to read a single byte from an I/O address port and *outb()* to write a single byte. The functions *inw()* and *outw()* manipulate 16-bit words, and *inl()* and *outl()* move 32-bit words ("longs"). The functions *repinsb()*, *repinsw()*, and *repinsd()* input a stream of bytes, 16-bit words, and 32-bit words, respectively, from an I/O port to kernel memory. The functions *repoutsb()*, *repoutsw()*, and *repoutsd()* output streams of bytes, 16-bit words, and 32-bit words, respectively, from an I/O port to kernel memory. The syntax of these function calls is shown below, and some of the calls are used in the drivers shown in Section "Device driver examples".

```
unsigned□char□inb(port)
int□port;
outb(port,□data)
int□port;
char□data;
unsigned□short□inw(port)
int□port;
outw(port,□data)
int□port;
short□data;
long□inl(port)
int□port;
outl(port,□data)
int□port;
long□data;
repinsb(port,□addr,□cnt)
int□port,□cnt;
char□*addr;
repinsw(port,□addr,□cnt)
int□port,□cnt;
short□*addr;
repinsd(port,□addr,□cnt)
int□port,□cnt;
long□*addr;
repoutsb(port,□addr,□cnt)
int□port,□cnt;
char□*addr;
repoutsw(port,□addr,□cnt)
int□port,□cnt;
short□*addr;
repoutsd(port,□addr,□cnt)
int□port,□cnt;
long□*addr;
```

As described earlier, with raw devices it is the driver's job to copy this data between the kernel's address space and the user program's address space whenever the user makes a *read()* or *write()* system call.

5.3.9 DMA allocation routines

A DMA controller has control registers defining the DMA start address and word count that the driver must manipulate. (See Section "Sharing interrupts and DMA channels" later in this section.) These routines allow DMA usage to be locked against DMA requests by other drivers. Not all devices use DMA, but those that do must have exclusive access to their DMA channel for the duration of the transfer.

The number of DMA channels is hardware-dependent. Some channels are reserved for such invisible housekeeping functions as screen refresh and cannot be reallocated. The names of the various channels are defined in the file *dma.h*.

Some machines have DMA chips that malfunction when more than one allocated channel is used simultaneously. To allow installation on these machines, the *dma_single* flag is set by default. On machines that do not have this deficiency, clear the *dma_single* flag to allow simultaneous DMA on multiple channels. This can be done by using the *idtune(1M)* command (see [7]) to set DMAEXCL to 0 (legal values are 0 and 1).

The DMA allocation routines and their parameters are described in [16]:

- Bus Master DMA,
- 8237 Onboard Chip DMA.

5.4 Reliant UNIX system driver specifics

5.4.1 Types of device

There are two classes of devices: block and character (or raw). As the term "block device" implies, the data on the device is formatted and addressed in "blocks." The terms "character device" and "raw device" imply that the kernel reads the data raw or unbuffered (except in the case of STREAMS). Some devices can be both block and character devices, implying that the system can access the device in two ways.

Although device drivers are normally associated with real devices, some drivers may have no hardware counterpart. These "devices" are often referred to as *pseudo* devices. For example, a trace driver may log certain classes of events. User programs write to the driver to record the events and read from the driver to recall the information. The trace driver would have internal mechanisms for formatting and storing the data. No hardware is associated with the driver, and the driver interfaces with software only. The Section "The trace driver" contains a sample trace driver as a device driver model. You may actually use this driver to help debug the driver you are developing.

5.4.2 Special files

The Reliant UNIX System treats a device as if it were a file; that is, when a user program wishes to access a device, it accesses the file that is associated with that device. These special files are sometimes called *nodes* or *device nodes*. The system calls that access regular Reliant UNIX System files (such as */etc/passwd*) are the same calls that access devices (such as */dev/console*). The system calls are *open()*, *close()*, *read()*, *write()*, and *ioctl()*. In Section "Function specifications (driver entry points)", the system calls at the driver level are described in detail.

5.4.3 Major and minor numbers

The device major numbers are used by the system to determine which device driver to execute when a user reads or writes to/from the special file. The system maintains two tables for mapping I/O requests to the drivers: one table for "character special" and the other for "block special." This implies that there are two sets of major numbers, one for character devices and one for block devices. Both start at zero and are numbered up to the last used major number. If you do an *ls -l /dev*, you may find that two very different devices have the same major number. That's probably because one is a "block special," using the block major number, and the other is "character special," using the character major number. For those drivers that are both block and character devices, such as the floppy driver, one major number of each type must be assigned. In this case, the actual numbers may be different and, in fact, often are different.

The minor number is entirely under control of the driver writer and usually refers to "subdevices" of the device. These subdevices may be separate units attached to a controller. A disk device driver, for example, may talk to a hardware controller (the device) to which several disk drives (subdevices) may be attached. The Reliant UNIX System accesses different subdevices using the different minor numbers.

5.4.4 The /dev directory

By convention, all device files are contained in the directory */dev*. The names of the files are generally derived from the names of the hardware, a convention that allows users to know what the device is by looking at the file name. For example, it would be confusing if the file */dev/tty* were a disk. Part of the name of the device file usually corresponds to the unit number of the device to be accessed via the file or, specifically, the minor number.

A new convention of Reliant UNIX 5.43 and other Reliant UNIX Systems is that */dev* can contain subdirectories that hold the nodes for all the subdevices of a particular type. This reduces the clutter in the */dev* directory. For example, */dev/dsk* contains all the "block special" files for the floppy and hard disks; */dev/rdsk* contains all the "character special" files.

The device file may exist in the file system even though the device is not configured in the running system. If a user attempts to access the device, or more specifically, the file, an error will result on the system call. Conversely, the device may be configured into the running operating system without the device file in the file system, in which case the device is inaccessible.

5.4.5 The master and system files

Associated with device drivers are two device configuration files: the Master file and the System file. For Reliant UNIX 5.43, the device driver portions of the traditional master file are in a file named *mdevice*. The device driver portions of the system files are in a file called *sdevice*. See [7] and *mdevice(4)* and *sdevice(4)* in [9] for information describing the *mdevice* and *sdevice* file format.

The *mdevice* file contains the device name (15 characters or less), the definition of the functions the device supports (second column has an "r" if the *read* function is implemented, has a "w" if *write* is implemented, and so forth), the block and/or character major number, and other descriptive information about the driver.

The *sdevice* file contains information on whether and how the device is installed in the system, that is, the number of units (subdevices), interrupt vector number (IVN) used, and other local information.

5.4.6 Structure of the device driver source files

Include files

Every file in the operating system source code includes header files containing declarations of global data structures. The source code for device drivers need not be contained in a single file, and programmers should subdivide the driver among several files if it is large. Even if the driver is contained in a single file, programmers should follow convention and declare the driver data structures in new driver-specific header (".h") files. The definition of the data structures (the place in the source code where the compiler allocates memory storage) should be in a ".c" file, usually the driver source file. The only data structures that should be defined outside the driver are those that are configuration-dependent; that is, if the driver needs to allocate storage for each subdevice, a method is needed to allocate based on the number configured. For Reliant UNIX 5.43, the file *Space.c* is used to allocate configuration-dependent data for use by the device driver.

For instance, if a system is configured for 4 trace devices, the file *Space.c* will include a line

struct trace tr_data[TR_UNITS];

and the include file for the trace driver will contain the declaration of the *trace* structure. The configuration process will set TR_UNITS equal to 4 based on the *unit* parameter (column 3) of the System file.

The driver source code file should "include" the new header files. Driver file names conventionally contain the device name as part of their names.

As an example, consider a driver for a new networking device called *nnet*. Assume the driver consists of two ".c" files, *nnet.c* and *nnetprot.c*, and one header file, *nnet.h*. The names suggest that the files are associated with the new *nnet* device and that the *nnetprot.c* file contains a protocol for the device. The header file may

contain a declaration such as

```
struct nnet {
        char    nn_state;
        char    nn_flags;
        int     nn_port;
        int     nn_chan;
        struct  nn_queue    *nn_qptr;
};
```

and the ".c" files should contain the line

```
#include        "sys/nnet.h"
```

General system data structures

Driver programmers must not change standard system header files, such as the *proc* file, the *user* file, or the *inode* file. Since the drivers are a separate part of the system, it is unacceptable to introduce new data structures and new "hooks" into standard system data structures to accommodate a private driver. In addition, changing system data structures could cause user-level programs to work incorrectly if they rely on the system data structure.

Usually driver source code must contain some standard "include" files to allow the driver access to system utilities and data structures commonly used to return information to the kernel. The list below defines a few of the more commonly used include files:

/usr/include/sys/types.h
: basic system data types

/usr/include/sys/param.h
: fundamental system parameters

/usr/include/sys/signal.h
: definition of system signals If the driver sends signals to user processes, it must include this file.

/usr/include/sys/conf.h
: definition of device switch tables This file is needed for the driver to define its *devflag* value.

/usr/include/sys/file.h
: definition of file structure This file is needed if the driver uses control flags such as "no delay" (FNDELAY).

/usr/include/sys/buf.h
: definition of the *buf* (system buffer) structure This file is needed if the driver uses the system buffer pool (see the section "Buffer pool").

/usr/include/sys/kmem.h
: defines for using the Kernel Memory Allocator This file is needed if the driver allocates memory for buffers out of the common memory pool.

/usr/include/sys/ddi.h
: Defines for using Device Driver Interface(DDI) routines. Note that this should be the last header file in the list of included header files.

5.4.7 Driver-specific data structures

Naming conventions

The names of driver data structures and variables should have the driver name in the prefix to ease program readability and debugging and to avoid conflict with other variables in the system with the same name. For example, in Section "The trace driver", the trace driver contains the variable *tr_cnt* and the data structure *tr_data*. Both names are private to the trace driver, and the prefix "tr_" identifies them as belonging to the trace driver.

Unit numbers

As mentioned above, drivers frequently control several hardware units, as a terminal driver may "drive" many

terminals. Each terminal has a unit number corresponding to the minor number of the device file. Drivers typically contain a data structure that contains a flag field to record the device status, such as open, sleeping, waiting for data to drain, and so forth. Except for the inclusion of a flag field, the contents of the data structure are device-dependent, so no recommendation can be given here. However, there should be one entry per unit, defined in the driver file and declared in the header file. A sample declaration of the data structure for the fake device *nnet* was defined above. Each *nnet* device should have one of these data structures.

devflag

Each driver should define a *devflag* variable so that the kernel knows the characteristics of the driver. The format is:

int nnetdevflag = val;

val may be a combination of flags. Each flag defines a special feature of the driver. For example, D_DMA should be set if the driver does DMA. If no flags are needed, *val* should be 0. The different flag values are defined in *usr/include/sys/conf.h*.

```
/*□Device□flags.*/
#define□D_NEW□□□0x00□□□□/*□New-style□driver□*/
#define□D_OLD□□□0x01□□□□/*□Old-style□driver□*/
#define□D_NPRIV□0x02□□□□/*□Driver□does□not□allow□private□access□*/
```

For compatibility, ID/TP recognizes all SVR 3.2x based block device drivers as "old-style" drivers. A "new-style" driver is recognized as such only if the 'characteristic field' (third field) of the *mdevice* entry contains character 'f'. This convention is new for the Reliant UNIX 5.43, and signifies that the driver had defined *devflag* variable. The d_flag field for the older type of block device is set by the ID/TP to point to a kernel defined integer variable *nodevflag* which the kernel initializes as D_OLD.

5.5 Function specifications (driver entry points)

This section describes the functions that form the driver interfaces to the kernel. For a raw device, they are *init*, *start*, *open*, *close*, *read*, *write*, *ioctl*, *chpoll*, and *halt*. For a block device, they are *init*, *start*, *halt*, *open*, *close*, and *strategy*. A driver need not contain every routine if one or more are irrelevant (a line printer driver usually does not have a read routine). If a device is both raw and block, the driver must contain all the functions, as appropriate. All the above routines are documented extensively in [16]. You are advised to refer to it for more information on these routines. Another point to be taken note of is, there are additional entry points available to driver writer that are specific to Reliant UNIX 5.43. They should be only used if absolutely necessary.

5.5.1 Poll

The routine *nnetpoll*, if present, is called by the system clock during every clock tick. It is useful for repriming devices that may lose interrupts.

nnetpoll(int ps)

The parameter *ps* is an integer that indicates the previous process's priority when it was interrupted by the system clock.

5.5.2 Kenter

The routine *nnetkenter*, if present, is called whenever the kernel is entered from user mode. It should be used with extreme care. Significant overhead in a *kenter* routine could adversely affect system performance.

nnetkenter()

The *kenter* routine is called with interrupts disabled. It must not enable interrupts, use *spl* routines, use *printf* statements, or *sleep*.

5.5.3 Kexit

The routine *nnetkexit*, if present, is called whenever the kernel is about to return to user mode. It should be used with extreme care. Significant overhead in a *kexit* routine could adversely affect system performance.

nnetkexit()

The *kexit* routine is called with interrupts disabled. It must not enable interrupts, use *spl* routines, use *printf* statements, or *sleep*.

5.5.4 Interrupt handler

As described earlier, hardware interrupts cause the processor to stop its current execution stream and to start executing an instruction stream that services the interrupt. The system identifies the device causing the interrupt and accesses a table of interrupt vectors to transfer control to the interrupt handler for the device.

The exact mechanism of associating interrupt vectors with interrupt handlers varies on different Reliant UNIX Systems. The discussion here assumes the system finds the correct interrupt routine on receipt of the device interrupt, and it assumes that the system executes the interrupt routine at a processor execution level high enough to prevent more interrupts of that type. For Reliant UNIX 5.43, there are a limited number of available interrupts. For more information on this and other machine-dependent aspects of the Reliant UNIX 5.43 interrupt architecture, see the section "Interrupts".

The device interrupt handler routines handle device interrupts, which are the device response to data transfers and requests. System software cannot predict when a device will interrupt the system. Typically, a system call blocks, that is, sleeps on an event, awaiting the device to interrupt. The device interrupt causes the system to invoke the interrupt handler that, in turn, awakens the blocked system call. For instance, device open routines may block until the device interrupts and "announces" its connection; or device read routines may block until the device interrupts and "announces" that data has arrived and can be read into the system.

Upon receipt of the interrupt, the kernel calls the driver interrupt handler:

nnetintr(int ivn)

where *ivn* indicates the interrupt number (hardware-dependent) associated with the interrupt, which is determined on the particular controller board. The vector field in the *sdevice* file for that controller board must also contain the interrupt vector number.

If the system is configured with two peripheral interrupt controllers (PICs), ivn can be 0, 1,3-15, depending on the hardware. The values reflect the 15 available interrupt lines on the two PICs combined. (Interrupt vector 2 is unavailable because it is used to wire the second PIC to the first PIC.)

The *ivn* argument can be used to determine which controller interrupted, in case where the driver supports multiple instances of a controller (each controller set at a different *ivn*). Interpretation of the *ivn* arguments is machine-dependent.

The interrupt handler must identify the reason for the interrupt (device connect, write acknowledge, data available) and set or clear device state bits as appropriate. It can also awaken processes that are sleeping (see Section "Sleep and wakeup"), waiting for the event corresponding to the interrupt. Interrupt handlers must not set any fields in the u area, particularly *u.u_error*, because the interrupted process is independent of the interrupt. For the same reason, interrupt handlers must not call *sleep()*.

5.5.5 Sharing interrupts and DMA channels

The Reliant UNIX 5.43 Installable Driver (ID) scheme allows for the sharing of interrupt lines and DMA channels among device drivers. When an interrupt occurs, then depending on the machine it may be that the interrupt handler for each device sharing the interrupt will be called. Each interrupt routine must first poll its device to see if the interrupt belongs to them. If not, they must return immediately with no processing so that the correct interrupt routine can execute.

The default during kernel configuration is to disallow devices to share interrupts. This prevents inadvertent re-use of interrupts or new drivers from sharing interrupts with old drivers expecting the interrupt to themselves. To indicate that a device can share its interrupt, column 5 of the *sdevice* (*type* field) entry must include a *3*. All devices sharing this interrupt must also have a *3* in this field. If they do not, an error will result during kernel configuration. See [7] and [9] for manual pages describing the *sdevice* file format.

To indicate that a device can share its DMA channel, column 3 of the *mdevice* (the *characteristic* field) entry must include a *D* identifier. All drivers sharing DMA channels must include a *D* in column 3 of their *mdevice* entry. If they do not, an error will result during kernel configuration. See [7] and [9] for manual pages

describing the *mdevice* file format.

5.5.6 Function naming conventions

The names of the driver *open*, *close*, *read*, *write*, *ioctl*, *strategy*, *init*, *poll*, *halt*, and *interrupt* routines must be prefaced by the generic driver name. For example, the names of the routines for the *nnet* driver are *nnetopen()*, *nnetclose()*, *nnetread()*, *nnetwrite()*, *nnetioctl()*, and *nnetintr()*. There are no restrictions on names for other functions in the driver, but it is best to preface the function names with the driver name for identification purposes, so you do not mistakenly define a function already defined in other parts of the operating system.

5.6 Kernel functions

The driver calls kernel routines to perform system-level functions, many of which were introduced in Section "Driver activities and responsibilities". The following paragraphs describe the syntax and use of these kernel functions.

5.6.1 Sleep and wakeup

As described in Section "Sleeping and waking processes", drivers must sometimes suspend or block their execution to await certain events, where an event is a system state in hardware or software. The driver waits by calling the sleep function, and the system schedules another process (context switch).

The sleep function takes two parameters: the identifier referencing an event upon which the process will sleep and a priority value that is assigned to the process when it is awakened:

```
sleep(caddr_t addr, int pri)
```

The identifier used for sleeping is an arbitrary value that has no meaning except to the corresponding *wakeup()* function call. The sleep addresses are usually taken from the entry in the device data structure of the device the process is accessing to guarantee uniqueness across the system. When a process goes to sleep awaiting an event, the driver should set a flag in the device data structure indicating the reason to sleep:

```
driver.state |= condition;
sleep(&driver.state, PRIORITY);
```

Later, either an interrupt handler or another process will call the *wakeup()* function to awaken the sleeping process. The code invoking the *wakeup()* function should check for a particular flag bit, indicating the reason that the process is sleeping. The driver then calls *wakeup()* with one parameter, namely the address where a process could be sleeping.

```
wakeup(caddr_t addr)
wakeup(&driver.state)
```

It is best for code readability and for efficiency to have a one-to-one correspondence between events and sleep addresses; one address should not be used for sleeping for two events. Again for clarity, there should be one bit in the flag field corresponding to every sleep event and, hence, to every sleep address. The *wakeup()* function awakens all processes sleeping on the address, enabling them to execute when the scheduler chooses them.

Therefore on return from *sleep()* it is generally necessary to re-test the condition which put the process to sleep.

```
while (locked) {
        sleep((locked), PRI);
}
```

If no process is sleeping on the address when *wakeup()* is called, *wakeup()* returns with no bad side effects.

It is illegal to call *sleep* when handling an interrupt since a process independent of the device could have been executing when the device interrupted. If the interrupt handler goes to sleep, the process that was interrupted is effectively put to sleep for reasons beyond its control. But second and far more important, sleeping in an interrupt handler could cause the system to crash in some Reliant UNIX System implementations because of the interdependency of the process context switch mechanism and interrupt levels. The interrupt handler must, therefore, not invoke other functions that could lead to a call to *sleep()*.

5.6.2 Delay function

This function is used to stop execution of the current process for a given period of time. Drivers can use the *delay* function instead of the *timeout* function, to instruct the driver to sleep for a specified amount of time and then wakeup. To use *delay*, specify the amount of time to wait. *delay* automatically calls *wakeup* via the *timeout* mechanism to resume execution.

The following piece of code illustrates the use of *delay* This code is from a driver for a line printer. Before allocating buffers and storing data in them, the driver checks the status of the device. If the printer needs to have paper loaded, it displays a message on the system console. If the driver called sleep directly, the operator would have to signal when the paper loaded. By using *delay*, the driver waits one minute and tries again. If paper was loaded, processing will resume automatically.

```
while (rp->status & NOPAPER) /* while printer is out of paper */
{                  /* display message & ring bell on system console */
    cmn_err(CE_WARN, "xx_write: NO PAPER in printer %d 07",(dev&0xf));
    delay(60 * HZ);       /* wait one minute & try again */
}       /* endwhile */
```

5.6.3 Block driver biowait/biodone event synchronization

Block-access drivers using the buffer header scheme that are waiting for an I/O event use *biowait/biodone* pair instead of *sleep* and *wakeup*.

The *biowait* function can be used to block a process until the I/O operation is complete. *biowait* sleeps at a priority of 20(*PRIBIO*). Since it operates on an I/O buffer header, it is not used by a character device(although it is used by a block devices doing raw I/O through *physio*).

In Reliant UNIX 5.43, *biowait* sets the *B_WANTED* flag. In addition, *brelse* is called by the *biodone* function.

5.6.4 Setting processor priority levels

As described in Section "Critical sections of the driver", the system allows devices to interrupt the processor and handles the interrupts immediately. The integrity of system data structures could be destroyed if an interrupt handler were to manipulate the same data structures as a process executing in the driver.

To prevent such problems, the system has special functions that set an interrupt lock to prohibit interrupts below certain levels. The functions are *splN*, where *N* ranges between 0 and 7 and corresponds to the priority level that it has in the kernel. *spl0()* allows all interrupts to occur, and *spl7()* allows none. Most Reliant UNIX Systems have an *splhi()* function to set the processor execution level to the highest value, which is *spl7()* for Reliant UNIX 5.43.

All *spl* functions return the previous priority level of the parameter passed to it. The *splx()* function is useful in cases where the processor priority level may have been raised already but where the driver does not know that it has been raised sufficiently to block out the proper level of interrupts. When the driver is ready to lower the priority level, it should not lower it all the way to 0 but rather to the old priority level. Consider the following code:

```
register int s;
.
.
s = spl5();
/* critical section of code */
splx(s);
```

Particulary nasty race conditions can occur if *spl* functions are not used with the *sleep()* function. For example, the code segment

```
driver.state |= condition;
while (driver.state & condition)
        sleep(&driver.state, PRIORITY);
```

will cause the process to sleep if the condition bit is set in the field *driver.state*. (Since processes could sleep on the address for several events, the sleep call is enclosed in the *while* loop so that when awakened, the code will again check that the condition is indeed no longer true. This is one reason it is best to sleep on

different address values for different sleep reasons.) Without use of the *spl()* function, the process could check the condition bit, find it true, and attempt to call sleep. But if an interrupt occurred before the process called sleep and the interrupt handler checked the condition bit to determine if a process was sleeping, it would assume the process was asleep and call *wakeup* to awaken it. Consider the following code:

```
if (driver.state & condition)
{
      driver.state &= ~condition;
      wakeup(&driver.state);
}
```

By the time the interrupted process calls *sleep()*, it will have missed the *wakeup()* call, and another one may never come. By bracketing the calls to *sleep()* with *spl()* function calls, the driver prevents the race condition:

```
int oldpri;
oldpri = spl5();
driver.state |= condition;
while (driver.state & condition)
       sleep(&driver.state, PRIORITY);
spl5(oldpri);
```

5.6.5 Interrupt priority level

Another kernel characteristic, Interrupt Priority Level (IPL), interacts with the *spl* functions. Some processor architectures have a hardware priority scheme that defines a hierarchy of which devices can interrupt others. Since the 80386 processor does not have such a scheme, Reliant UNIX 5.43 has assignable priority levels that simulate hardware priority levels. By defining an IPL in the *sdevice* file, we can protect a driver's critical regions at the appropriate level. IPL8 is the highest level and is reserved for the internal clock. Drivers at this level cannot be interrupted by other devices (their interrupt routines execute at *splhi*. A device at IPL6 can be interrupted by a device at IPL7 or IPL8. In Reliant UNIX 5.43, the base system device drivers use the following IPL levels. This shows that the serial ports run at the highest priority to prevent loss of data.

The line printer is more safely interrupted and is given a low IPL. See Section "Controller interface basics", for a more complete definition of the device configuration assignments.

DEV	IPL	Device attached
clock	8	Reliant UNIX System clock
asy	7	Serial Ports
fd	6	Floppy Disk
hd	5	Hard Disk
kd	6	Keyboard
lp	3	Line printer (Parallel Port)
rtc	5	Real Time Clock

Table 34: Interrupt priority levels for attached devices

Care must be taken to limit the amount of time spent at high levels. For example, if any driver elevates to *splhi* for more than a few milliseconds, loss of Reliant UNIX System clock time may result.

5.6.6 Sleep priorities

The second parameter to the *sleep()* function, a scheduling parameter, is used when the process awakens from its sleep. The parameter, called the sleep priority, has critical effects on the sleeping process's reaction to signals. Thus a low sleep priority watches for a higher process priority after wakeup. If the sleep priority is lower than the manifest constant PZERO (25 on most systems), then the system does not awaken sleeping processes on receipt of a signal. However, if it is lower than PZERO, then the system awakens sleeping processes "prematurely." If the PCATCH bit (discussed later) is not set, the process immediately finishes the

system call, that is, it executes a *longjmp()* out of the driver.

Sleep calls the *longjmp()* function. When the system executes *longjmp()*, it does not follow the conventional C function call/return sequence but instead resets the program counter, stack pointer, and data registers to the values they had when the most recent *setjmp()* function call was done.

For instance, if a signal is sent to a process sleeping in the following sleep call, the system call will end immediately without returning to the code that called sleep:

```
sleep((caddr_t)&tp->t_rawq, PZERO + 5);
```

When a driver must call sleep, how should the driver programmer determine the sleep priority? The first decision is whether the process should ignore the receipt of signals or not. If the driver puts the process to sleep for an event that is "sure" to happen, then it should ignore receipt of signals and sleep at a priority less than PZERO.

An example of an event that is "sure" to happen is waiting for a locked data structure to be unlocked:

```
if (tp->t_state & T_LOCKED)
        sleep(&tp->t_state, PZERO - 5);
```

In this case, another process locked the data structure and went to sleep, but it left the data structure locked so that no other process could change it before it awakened. Since that process will eventually awaken and unlock the data structure and then awaken all other processes waiting for the lock to clear, the event (the *wakeup* call announcing the unlock) is sure to happen. Otherwise, the driver has a bug.

If the driver puts a process to sleep while it awaits an event that may not happen, the process must sleep at a priority greater than PZERO. An example of an event that may not happen is waiting for data to arrive from a remote device. For example, when the system reads data from a terminal, the *read* system call sleeps in the terminal driver waiting for data to arrive from the terminal. If data never arrives, the read will sleep indefinitely. When a user at the terminal hits the <BREAK> key or even hangs up, the terminal driver interrupt handler sends a signal to the reading process still asleep, and the signal causes the reading process to finish the system call without having read any data. If the driver had slept at a priority value that ignores signals, the process could have been awakened only by a specific *wakeup* call. If that *wakeup* call could never happen (the user hung up the terminal), then the process would sleep forever, clearly an undesirable characteristic.

When the driver programmer decides whether the process should ignore signals or not, he/she must choose the priority values so as not to affect process scheduling adversely. The system should be benchmarked using several sleep priority values to tune system performance with the new driver.

Drivers must occasionally "clean up" on receipt of a signal while sleeping before returning to upper levels. Since the *longjmp()*, as discussed so far, takes place directly from the sleep function call, the priority parameter to the sleep function call has additional meaning: if the priority parameter is or'ed with the manifest constant PCATCH, the sleep call returns the value 1 if awakened on receipt of a signal. But if the sleeping process is awakened by an explicit *wakeup* call rather than by a signal, then the sleep call returns 0. The following code sequence allows the driver to clean up before returning.

```
if (sleep(sleep_address, condition | PCATCH))
{
        /* driver code cleanup */
        .
        .
        .
        return(EINTR);
}
```

Typical items that need cleaning up are locked data structures that should be unlocked when the system call completes.

```
tp->t_state |= TLOCK;   /* locks the driver unit */
tp->t_state |= TSLEEP;
if (sleep((caddr_t) &tp->t_state, TPRI | PCATCH))
{
```

```
□□□□□□□□□□□□tp->t_state□&=□~(TLOCK□|□TSLEEP);
□□□□□□□□□□□□return(EINTR);
}
/*□somebody□woke□up□driver...
□*□continue□normally□here□*/
```

5.6.7 Timeout

Sometimes, a driver arrives at a state where it wishes to re-enter itself after a specified time. The driver uses the *timeout()* function for this purpose. Timeout takes three parameters: the function to be invoked when the time increment expires, the value of a parameter with which the function should be called, and the number of clock cycles to wait before the function is called. A sample *timeout()* call is

```
id = timeout(repeat, n, count);
```

where *n* is the parameter to the function *repeat()*, to be called after *count* cycles. If *count* is 100 and if the clock interrupts the processor 100 (defined by the parameter *HZ* in */usr/include/sys/param.h*) times a second, the system will execute the function *repeat()* in 1-second real time as a result of the above *timeout()* call.

The exact time until the timeout takes effect may not be precise because of the interaction of other parts of the system. The compiler requires prior declaration of the function name parameter to *timeout*, as in

```
extern□char□*repeat();
.
.
.
timeout(repeat,□n,□count);
```

depending on where *repeat()* is defined.

5.6.8 Error reporting

One of the most important aspects of writing a device driver is the correct handling of errors. Driver code must handle any error condition, or the consequences may be severe. For example, a stray interrupt should be a trivial event, but could crash the system if the driver is not prepared to handle it. The crash could cause data corruption and physically damage the system. When an error occurs, the driver can do one or more of the following:

- Write the error condition to a structure so the driver knows about it. At process level the error is placed in a state structure and returned to the caller. At the interrupt or base level, errors on block devices can be recorded in the *b_error* member of the *buf* structure.
- Retry the process. The errror may be a transient problem. Some hardware device boards have retry capabilities;let these boards do the the retry. But if the error is software related, the driver must decide how many times to retry.
- Report the error to a system error log. If the error is severe, take the faulty hardware out of service to minimize the damage and keep the system running normally.
- Report the error to the system administrator, either by printing it on the system console, and or by writing it to *putbuf* (see *cmn_err()* in Section "Kernel print statements").
- Send a signal to a user process.

5.7 Dynamic memory allocation

The routines that allocate data space from memory for internal operating system use are machine-dependent and beyond the scope of this document.

5.7.1 Allocating buffer space

As mentioned in the discussion on the driver *read()* and *write()* routines, drivers may require buffers for passing data around. The following utility routines in the Reliant UNIX System provide buffer space.

Kernel memory allocator

The Reliant UNIX System V/386 Release 4.0 Version 1.0 provides routines to allocate and release kernel memory, which can be used by drivers. Refer to[16] for more information on these routines.

Buffer pool

The Reliant UNIX System provides a set of buffers that are normally used for file system I/O, but they can be "borrowed" by drivers if they follow the rules outlined here. The driver must include the header file *sys/buf.h*. The functions that drivers may use to manipulate the buffers are

- brelse(bp) struct buf *bp;

  Releases a previously allocated buffer.
- biowait(bp) struct buf *bp;

  Sleeps on the buffer awaiting an event, such as completion of I/O.
- biodone(bp) struct buf *bp;

  Awakens a process sleeping via *biowait()*.
- clrbuf(bp) struct buf *bp;

  Clears the contents of the buffer (sets every byte in the buffer to 0) whose header is the pointer bp.

The driver may access the buffer header field *b_flags* to access buffer state flags and the field *b_un.b_addr* to get the address where the data buffer resides. Acceptable flags to use in the *b_flags* field include

B_WRITE
: when writing data from the buffer to the device.

B_READ
: when reading data from the device.

B_DONE
: set by the function *biodone()*, to indicate that the I/O operation has completed.

B_ERROR
: to indicate an error in use of the buffer.

B_BUSY
: to lock the buffer and prevent other processes from accessing the buffer. Use of the B_BUSY flag prevents other processes from accessing the buffer if they first check the flag to see if it is busy.

```
while (bp->b_flags & B_BUSY)
    sleep(bp, DRIPRI);
bp->b_flags |= B_BUSY;
```

B_WANTED
: to indicate that a process is sleeping awaiting the buffer. The function *brelse()* clears the flags B_WANTED and B_BUSY, and the function *geteblk()* sets the B_BUSY flag. It is best to "or" and "and" the flags in, rather than just setting them.

Here is an example of the use of buffers in a tape driver:

```
tapecntl(dev, flag, opcode, arg1, arg2)
{
    register struct buf *bp;
    register int rcode;
    bp = (struct buf *) geteblk();
    /* CNTL flag is used to indicate this is a control buffer */
    bp->b_flags |= B_CNTL;
    /* set async flag so buffer will be released */
    if (flag == FNDELAY)
        bp->b_flags |= B_ASYNC;
    bp->b_dev = (MT0<<8)|dev;
    tapestrategy(bp);
    rcode = 0;
```

```
□□□□if(flag□!=□FNDELAY)□{
□□□□□□□□biowait(bp);
□□□□□□□□if(bp->b_flags&B_ERROR)
□□□□□□□□□□□□rcode□=□-1;
□□□□□□□□bp->b_flags□&=□~B_CNTL;
□□□□□□□□biodone(bp);
□□□□}
□□□□return(rcode);
}
```

5.7.2 Installable driver implementation

This section describes the Reliant UNIX 5.43 Installable Driver (ID) scheme, which allows users to add drivers for peripheral devices to their systems. This section provides an overview of what software developers are required to do when building an installable device driver package.

ID overview

The ID provides an automatic method of installing device drivers using the *pkgadd* command delivered in the Base System Package of the Reliant UNIX 5.43 Foundation Set. Driver developers must use the C Programming Language Utilities (CPLU) delivered in the C Software Development Set of the Reliant UNIX 5.43 Software Development Set to compile their driver and build installation scripts for delivery with the device driver (floppy diskette) package. In Section "Device driver development methodology", step-by-step procedures on how to write, compile, debug, and finally package the device driver are provided.

Users will be exercising ID when adding new driver(s) to their system. Performing ID is usually referred to as system reconfiguration and in the past has required users to know the internals of many system files (*/etc/system*, */etc/master*, *io.mk*, *space.h*, the *config* command, etc.). The ID builds a new Reliant UNIX System , then has the user reboot the system using the new kernel as is currently done on many other Reliant UNIX System implementations.

The ID provides a packaging strategy applicable to vendor-supplied drivers. Driver writers must develop an add-on driver software package (DSP) similar to those for applications programs. The DSP will consist of a driver object module, installation and removal scripts, and device-specific entries for system configuration, initialization and shutdown files, as well as space allocation entries normally associated with "*space.h*" on earlier Reliant UNIX Systems. The ID allows replacement of "base" drivers via a special DSP called an Update Package(UDSP). Base drivers are defined as those drivers delivered with the Reliant UNIX 5.43 Base System Set software.

5.8 Controller interface basics

I/O devices connect to controllers that are either resident on the 386 parent board or on a peripheral board. The controller interface generally requires

- an interrupt line designated by an interrupt vector number (IVN),
- DMA (if used by peripheral) channel number
- a port address range through which the CPU and device can communicate (IOA - I/O address), and
- an optional address range that references memory (usually dual-port RAM, as mentioned in the previous section) on the controller board (controller memory address (CMA)).

5.8.1 User interface

A user may install or remove device drivers using *pkgadd* and *pkgrm*. The *pkgadd* command installs a DSP onto Reliant UNIX 5.43 and initiates automatic procedures to reconfigure the kernel. The *pkgrm* command allows the user to select which package to delete. It then removes the DSP from Reliant UNIX 5.43 and reconfigures the kernel without the driver.

The *pkginfo* command displays any software packages that the user has installed. DSPs are treated identically to other Reliant UNIX 5.43 Software packages. Device drivers that are pre-installed on the system by the Base System Set floppy diskettes are not displayed by this command.

At this point, we assume a basic knowledge about the layout and implementation of OA&M packages.

User privileges

The DSP uses the same installation rules as any other Reliant UNIX 5.43 add-on software, that is, the user needs *root* permissions. A user must be working as system administrator to install DSPs.

Interactions with other Reliant UNIX 5.43 processes

The DSP affects other users or processes no more than installing or removing other software with the exception that the final step is to reboot the system. It is, therefore, not advisable for another user to be logged on via a remote terminal while installing or removing a DSP.

5.8.2 Number of installed drivers

Due to limited available interrupts, as defined in Section "Sharing interrupts and DMA channels", there is a limit to the number of conventional peripheral devices which can be installed on a Reliant UNIX 5.43. Additional drivers could, however, be installed for devices not requiring interrupts, for software pseudo-devices, or for devices sharing interrupts. (See Section "Sharing interrupts and DMA channels" above.)

5.9 Modifications for ID

5.9.1 Master file

In earlier Reliant UNIX Systems, the master file contained information about all I/O devices that can be configured into a kernel. It also listed tunable parameters and their default values. For Reliant UNIX 5.43, the master file has been split into

mdevice
: the master device file

mtune
: the master tunable parameter file

The format of *mdevice* and *mtune* are shown in [7] and in [9].

5.9.2 System file

The *system* file represents a configuration from which a kernel is configured. The system file has been split into

sdevice
: system device file

stune
: system tunable parameter file

sassign
: file that specifies the pseudo-devices *root*, *pipe*, *swap* and *dump*

The format of *sdevice* and *stune* are shown in [7] and in [9].

5.9.3 space.c

The amount of storage allocated for each driver data structure is dependent on the number of subdevices configured for a particular device. For Reliant UNIX 5.43, since there was a need to modularize storage allocation and since space allocation should rightly be done in a *.c* file, the file */usr/include/sys/space.h* has become a collection of *space.c* files stored in the */etc/conf/pack.d* directory.

These *space.c* files determine how much storage is required for the kernel and each of the added drivers and initialize other driver variables.

These files are compiled and linked into the kernel during reconfiguration.

5.9.4 ID directory structure

The root directory for the ID software is */etc/conf*. All files and directories are writable only by root so that users cannot inadvertently modify anything. The ID directory contains the following subdirectories:

bin
: Contains all ID commands.

cf.d
: Contains configuration-dependent files.

  stune/sassign/sdevice/mdevice/mtune
  : Equivalent to the master and system files of earlier Reliant UNIX Systems.The *mdevice* file is built from the Master modules of the installed DSPs. There is a "base" *mdevice* file supporting corresponding devices in base system. The entries in the Master modules for nstalled DSP's are added to the "base" *mdevice*.

  mfsys/sfsys
  : File system type information, see the *mfsys(4)* and *sfsys(4)* pages in [9]).

  init.base
  : The base system part of */etc/inittab*.

  kernmap
  : Kernel memory mapping information.

  Temporary files used by the reconfiguration process:

  conf.c
  : kernel data structures and function definitions

  config.h
  : kernel #defines for device and system parameters

  direct
  : listing of all driver components included in the build

  fsconf.c
  : File system type configuration data

  vector.c
  : Interrupt vector definition

  unix
  : The Reliant UNIX Operating System kernel; eventually to be linked to */stand/unix*.

  These temporary files are created and used by the ID reconfiguration software, then deleted. If you run the */etc/conf/bin/idconfig* command manually, it will create these files for your review.

sdevice.d
: Stores one file for each type of device (that is, controller board or pseudo-device). The file name will be the same as the DSP internal name. Each file contains all of the system configuration entries pertaining to that device. Generally, this file contains a single line entry. (A device might have multiple entries in the system configuration if there were two devices of that type installed in the system.) These files are copies of the *Systems* modules of each installed DSP. When concatenated together, these files comprise the file */etc/conf/cf.d/sdevice*.

pack.d
: Contains one directory for each DSP installed on the system. The directory name will be the same as the DSP internal name. The directories in *pack.d* contain the *Driver.o* and *space.c* files for the drivers. This directory can also contain a *stubs.c* file. *stubs.c* files are often used as "place holders" for references the kernel needs to resolve for code that has been uninstalled. These files are taken from the *Driver.o*, *space.c*, and *Stubs.c* files of a DSP. Note the change in capitalization for *Stubs.c* and *space.c*. A DSP must name these files starting with an uppercase letter. The ID tools will install the files into */etc/conf/pack.d* using the lowercase forms.

rc.d
: Contains startup procedures for each of the installed DSP's. There will be one file per device startup

procedure, and the file's contents is to be taken from the *Rc* module of the DSP. The names of the files will be the same as the DSP's internal names. The contents of this directory will be linked to */etc/idrc.d* whenever a newly configured kernel is first booted.

sd.d
: Contains shutdown procedures for each of the installed DSPs. There will be one file per device shutdown procedure, and the file's contents is to be taken from the *Shutdown* module of the DSP. The names of the files will be the same as the DSP's internal names. The contents of this directory will be linked to */etc/idsd.d* whenever a newly configured kernel is first booted.

node.d
: Contains device node definitions (special files in */dev*) for each of the installed DSPs. There will be one file per device driver, and the file's contents is taken from the *Node* module of the DSP. The file names will be the same as the DSP internal names. The contents of this directory is the input to the *idmknod* command.

init.d
: Contains */etc/inittab* entries for each of the installed DSPs. There will be one file per device driver, and the file's contents is taken from the *Init* module of the DSP. The file names will be the same as the DSP internal names. The contents of this directory is the input to the *idmkinit* command. (It should be noted that this directory may also contain */etc/inittab* entries other than those associated with DSPs.)

mfsys.d
: Stores one FS type master data file for each file system type add-on. These files are taken from the *Mfsys* module of a DSP. When concatenated together, these files comprise the file */etc/conf/cf.d/mfsys*.

sfsys.d
: Stores one FS type system data file for each file system type add-on. These files are taken from the *Sfsys* module of a DSP. When concatenated together, these files comprise the file */etc/conf/cf.d/sfsys*.

5.9.5 Device #defines generated by the configuration process

The configuration process produces a file *config.h* that contains device parameters in the form of *#define*s that specify the number of units, interrupt vectors used, and other pertinent information. For example, a device driver that controls several subdevices may not know how many subdevices are actually installed in the system but can determine the number by including *config.h* and referencing the proper *#define*. The parameters generated in *config.h* are prefixed with the device handler prefix in all capital letters as shown below:

#define PRFX	Set to 1 if device is configured.
#define PRFX_CNTLS	Number of entries in System (*sdevice*) file.
#define PRFX_UNITS	Number of subdevices (see below).
#define PRFX_CHAN	DMA channel used (-1 if none).
#define PRFX_TYPE	Interrupt vector type used.
#define PRFX_CMAJORS	Number of multiple Major Numbers supported.
#define PRFX_CMAJOR_O	Major Numbers supported. The first major is PRFX_CMAJOR_0, the second PRFX_CMAJOR_1 ,etc.

Table 35: Per-device defines

The following table lists the per-controller *#define*s (PRFX_0 represents the first controller, followed by PRFX_1, etc if more than one controller is installed):

#define PRFX_0	Set to 1 if controller 0 is configured.
#define PRFX_0_VECT	Interrupt vector used (0 through 15).
#define PRFX_0_SIOA	Starting Input/Output Address.

#define PRFX_0_EIOA	Ending Input/Output Address.
#define PRFX_0_SCMA	Starting Controller Memory Address.
#define PRFX_0_ECMA	Ending Controller Memory Address.

Table 36: Per-controller defines

It is important to note that since the device driver is delivered as an object module (*Driver.o*), the #define cannot be referenced therein. The correct way to access the value is in the DSP's *Space.c* file by defining a variable which is assigned the value of the #define. The driver object module can then simply reference the variable.

5.10 Commands for installing drivers and rebuilding the Reliant UNIX operating system kernel

The DSP *Install* script must use calls to *idcheck*, *idinstall*, and *idbuild*. Manual pages for these commands are provided in [7] and in [9].

5.10.1 Idcheck

This command is used to obtain selected information about the system configuration. The command is designed to determine if a particular driver package is already installed or to test for interrupt vectors, device addresses, or DMA controllers already in use. It is anticipated that it will be used in *Install* scripts that will test for usable IVN, IOA, and CMA values, then instruct the user to set particular switches or straps on the controller board.

5.10.2 Idinstall

The *idinstall* command is called by the DSP's *postinstall* and *preremove* scripts, and its function is to install, remove, or update a DSP.

5.10.3 Idbuild

The *idbuild* command is a shell script that comprises the reconfiguration processes.

- Concatenates the files in */etc/conf/sdevice.d* to produce the *sdevice* file.
- Concatenates the files in */etc/conf/mfsys.d* to produce the *mfsys* file.
- Concatenates the files in */etc/conf/sfsys.d* to produce the *sfsys* file.
- Executes the *idconfig* and *idmkunix* commands.
- Sets a lock file so that on the next system shutdown, */etc/conf/cf.d/unix* will be copied to */stand/unix*. On the next system reboot, the same lock file will enable the new driver configuration (nodes in */dev*, */etc/inittab*, etc.) to be installed.

The *idcheck* command does not work properly if *idbuild* has not been executed after a DSP has been added, deleted, or updated through the use of *idinstall*. In order to get around this, after executing *idinstall*, re-synchronize the *sdevice* file or, execute *idbuild*. If not desired, update the */etc/conf/cf.d/sdevice* file by doing:

```
cat /etc/confdevice.d/* > /etc/conf/cf.d/sdevice
sync
```

5.11 The driver software package

This section defines the contents of the Driver Software Package (DSP). Each DSP must have two "names." One is the "external name" that the user will see when the package is installed. The second is an "internal name" that the kernel uses to identify the device. More information is provided about these names below and in Section "Driver development procedures".

The DSP is to be delivered on an installation medium, as described in Section "Writing the floppy diskette". There you will find general descriptions of the files and information on ordering and contents. The driver writer must prepare a DSP consisting of the files (termed modules) described in the following sections.

The package should install the following files as class "volatile" in a tmp directory. Its *postinstall* script should

cd to that directory before invoking the ID commands to add the DSP to the system. An example OA&M *prototype* would be as follows:

```
#□packaging□files
i□pkginfo
i□postinstall
i□preremove
d□none□/tmp/foo□755□□□□□bin□□□□□bin
v□none□/tmp/foo/Driver.o=/usr/src/pkg/foo/Driver.o□?□?□?
v□none□/tmp/foo/Master=/usr/src/pkg/foo/Master□?□?□?
v□none□/tmp/foo/System=/usr/src/pkg/foo/System□?□?□?
v□none□/tmp/foo/Space.c=/usr/src/pkg/foo/Space.c□?□?□?
v□none□/tmp/foo/Rc=/usr/src/pkg/foo/Rc□?□?□?
v□none□/tmp/foo/Shutdown=/usr/src/pkg/foo/Shutdown□?□?□?
#□package□objects:
!default□555□bin□bin
d□none□□/usr/lib/foo□□□□755□□□□□root□□□□sys
f□none□□/usr/lib/foo/cmd=/usr/src/pkg/foo/cmd
f□none□□/usr/include/sys/foo.h=/usr/src/pkg/foo/foofblk.h□□□444□□bin□ bin
```

5.11.1 Driver.o (required)

This is the driver object module that is to be configured into the kernel. This object module must be compiled using the native C Programming Language Utilities (CPLU) delivered in the C Software Development Set of the Reliant UNIX 5.43 Software Development Set. In Section "Device driver development methodology" procedures for coding, compiling, and debugging the driver object module are provided.

5.11.2 Master (required)

This module contains a one-line description of the device being installed. This module will be added to the ID *mdevice* file. The syntax of this line appears in the *mdevice(4)* manual page in [7] and [9].

Columns 6 and 7 of the Master entry should be set to zero. These are the driver's character and block and character major device numbers. These values are set by ID when the Master entry is added to the kernel configuration. If the device needs to support more than 256 subdevices, multiple major numbers may be specified in either one or both of these fields.

The Reliant UNIX 5.43 supports multiple major numbers per device. In order to support a large number of subdevices, a new letter 'M' (upper case 'm') is used in the third column of the *mdevice* file.

An 'f' in the 3rd field identifies driver as "new-style" (based on DDI/DKI interfaces in this book). This applies to STREAMS, block and character drivers. If the driver defines *devflag* then the Master file must have an 'f' in the third field (see pg 3-12).

In order to specify the specific major numbers, a "range notation" is used that will specify a list of consecutive major numbers to be used. This notation will specify the first and last major numbers separated by a dash(e.g.,the range 3-6, will be interpreted as four major numbers between 3 and 6 inclusive). The fifth and sixth columns of the *mdevice* file (block and character major device numbers,respectively) may contain a range specification of majors. The implementation is backward compatible with all other *mdevice* entries that will continue to specify a single major number.

Notice the difference, in the following example, between the specification of single majors vs multiple majors.

Single major:

```
lp□□□□□□loc□□□□□iHcSf□□□□□□lp□□□□□□□□0□□□□□□□7□□□□□1□□□□□□□2□□□□-1
```

Multiple majors(fictitious device names used):

```
ft□□□□□□locrwi□□IHrobcfM□□□ft□□□□□□1-4□□□□□3-6□□□□□1□□□□□□□2□□□□-1
fg□□□□□□locrwi□□IHrbcfM□□□□fg□□□□□□□□5□□□□20-24□□□□1□□□□□□□2□□□□-1
```

The 'ft' entry specifies multiple majors for both block and character numbers while the 'fg' entry specifies a single block major and multiple character majors.

For devices that require that both the block and character major ranges be the same, a 'u'("unique") flag may be specified in the third column of *Master*. Devices that do not specify 'u' may be assigned different ranges for block and character majors.

5.11.3 System (required)

When a DSP is installed, this module is added to the files which will be included in the kernel the next time the system is rebuilt. During reconfiguration, the System modules for each device are concatenated together to form the ID file *sdevice*. The syntax of this line appears in the *sdevice(4)* manual page in [7] and [9].

5.11.4 Space.c (optional)

The amount of storage allocated for each driver data structure is dependent on the number of subdevices configured for a particular device. For Reliant UNIX 5.43 , each driver can have its own *Space.c* file containing configuration dependent-data structures. Each driver package brings in its own *space.c* file for space allocation.

As an alternative to providing *Space.c*, the driver writer could preallocate data in the driver, eliminating the need for this file. This is useful when

- the amount of storage required by the driver is static
- the difference in storage between the minimum and maximum number of subdevices that can be configured for that device is small

If the driver object file has been compiled with special *#ifdef*s turned on, it is important to explicitly turn on these *#ifdef*s in the *space.c* before including headers so that the compiled *space.c* uses the correct definitions of structures and types.

5.11.5 Node (optional)

This file is used to generate the device's "special files" in the */dev* directory on the next reboot after the system has been reconfigured. *Node* contains one line for each node that is to be inserted in */dev*. The columns can be separated by spaces. The syntax of this line is as follows:

Column 1:
DSP internal name

Column 2:
name of node to be inserted

Column 3:
'b' or 'c' (block or character device)

If the device supports multiple majors, a specific major can be specified via the following notation

b:offset or c:offset,

where *offset* is an offset number within the range of majors specified in the *mdevice*. This offset starts with '0' to specify the first number in the range. An offset of 1 would specify the second number in the range, etc.

Column 4:
minor device number

Example of a Node file

```
DSP-interner-Name□□node0□□c□□0
DSP-interner-Name□□node1□□c□□1
DSP-interner-Name□□node2□□c:0□0□□#□Wählt□1.□zeichenorient.□Ger.kl.nr. aus
DSP-interner-Name□□node3□□c:1□0□□#□Wählt□2.□zeichenorient.□Ger.kl.nr. aus
```

See the *idmknod(1M)* manual page in [7].

5.11.6 Init (optional)

Some drivers require entries in */etc/inittab* to make them operational. An *inittab* entry is of the following form

(see *inittab(4)* in [9]):

id:rstate:action:process

Each line of the *init* module must be of the format *action:process*, or *rstate:action:process*. The *id* and *rstate* field will be generated by ID (if your entry has an *rstate* field it will be used; otherwise, "2" will be used). The new *inittab* entries will be added to */etc/inittab* on the next reboot after the system has been reconfigured. For more information on the *init* module format, see the *idmkinit(1M)* manual page in [7].

5.11.7 Rc (optional)

This module is an initialization file that is executed when the system is booted. The new *Rc* file will be placed in the directory */etc/idrc.d* on the next reboot after the system has been reconfigured and will be invoked on every system reboot thereafter upon entering *init* level 2 (see the *init(1M)* manual page in [7]). When creating this module the file permissions must allow execution by *root*.

5.11.8 Shutdown (optional)

This file is executed when the system is shut down. The new *shutdown* file will be placed in the directory */etc/idsd.d* on the next reboot after the system has been reconfigured and will be invoked on every system shutdown thereafter upon entering *init* state 0, 5 and 6. When creating this module the file permissions must allow execution by *root*.

5.11.9 Postinstall (required)

This module requires

- Change directory to *tmp* directory where DSP files were installed.
- Use *idcheck*s to determine conflicts with the installed drivers.
- Invokes the ID command *idinstall* with the *-a* option and pass it one argument, the internal DSP name. This will move the contents of the DSP to the proper directories.
- Invokes the ID command *idbuild*.
- *removef* any */tmp* files installed.

The following is a sample *postinstall* script:

```
trap '1 2 3 5 15
#
#       386 Package Template
#       Driver files installed in /tmp/foo.
#       "foo" will be ID package name for the driver
#
TMP=/tmp/foo.ierr
ERROR1=" Errors have been written to the file $TMP."
rm -f $TMP > /dev/null 2>&1
PRFX=foo
cd /tmp/foo
/etc/conf/bin/idcheck -p ${PRFX} > /dev/null 2>&1
if [ $? != 0 ]
then
      echo "<PACKAGE NAME> has already been installed."
      exit 1
fi
echo "Installing the drivers."
/etc/conf/bin/idinstall -a ${PRFX} 2>> $TMP
if [ $? != 0 ]
then
      echo "\n\tThe installation cannot be completed due to an error
            in the <PACKAGE NAME> driver installation. $ERROR1 "
      exit 1
fi
/etc/conf/bin/idbuild 2>> $TMP
if [ $? != 0 ]
then
```

```
□□□□□□echo□"The□installation□cannot□be□completed□due□to□an□error
□□□□□□□□□□□□□in□the□driver□installation.□$ERROR1□"
□□□□□□/etc/conf/bin/idinstall□-d□${PRFX}□2>>/dev/null
□□□□□□exit□1
fi
installf□$PKGINST□/usr/options/$PKG.name
echo□$NAME□>□/usr/options/$PKG.name
installf□-f□$PKGINST
#□Needed□so□the□removef□works.
removef□$PKGINST□/tmp/foo/*□>□/dev/null□2>&1
removef□-f□$PKGINST
rm□-f□$TMP
exit□10
```

5.11.10 Preremove (required)

This module

- Calls the *idinstall* command with the *-d* option and passes it one argument, the internal DSP name. This will remove the DSP modules.
- Invokes the ID command *idbuild*.

The following is a sample *preremove* script:

```
trap□'□1□2□3□5□15
#
#□□□□□□□386□generic□driver□preremove
#□□□□□□□ID□package□prefix□is□foo
#
TMP=/tmp/foo.rerr
ERROR1="An□error□was□encountered□removing□the□<PACKAGE□NAME>□package.
□□□□□□The□file□$TMP□contains□errors□reported□by□the□system."
ERROR2="The□kernel□rebuild□failed.□□However□all□software□dealing□with
□□□□□□the□<PACKAGE□NAME>□has□been□removed.□□The□file□$TMP□contains
□□□□□□errors□reported□by□the□system."
rm□-f□$TMP□>□/dev/null□2>&1
PRFX=foo
/etc/conf/bin/idinstall□-d□${PRFX}□2>>□$TMP
if□[□$?□!=□0□]
then
□□□□□□echo□$ERROR1
□□□□□□exit□1
fi
/etc/conf/bin/idbuild□2>>□$TMP
if□[□$?□!=□0□]
then
□□□□□□echo□$ERROR2
□□□□□□exit□1
fi
rm□-f□$TMP
exit□10
```

5.11.11 Summary of modules

Module	Mandatory/ Optional	Definition
preremove	M	Remove DSP
postinstall	M	Install DSP driver files
Driver.o	M	Driver object file
prototype	M	OAM package *prototype* file
Master	M	Master file entry
System	M	System file entry

Space.c	O	Driver space allocation file
Stubs.c	O	Module deconfiguration
Node	O	Special file entries in */dev*
Init	O	*/etc/inittab* entries
Rc	O	Executed when entering *init* level 2
Shutdown	O	Executed when entering *init* level 0
Mfsys	O	File system type master data
Sfsys	O	File system type system data

Table 37: Components of driver software package

5.11.12 Base system drivers

An examination of */etc/conf/cf.d/mdevice* will show the installed DSPs. A partial list of the base system device drivers and software modules is as follows:

Hardware device drivers	Definition
asy	Serial ports driver
fd	Floppy disk
hd	Hard disk
kd	Keyboard
lp	Line printer (parallel port)
rtc	Real time clock

Table 38: Base system device drivers

Software modules	Definition
ipc	Interprocess communication (IPC)
ld0	TTY line disciplines
mem	Memory "driver"
msg	IPC messages
prf	Kernel profiler
sem	IPC semaphores
shm	IPC shared memory
sxt	Shell layers
xt	Layers

Table 39: Software modules

The above list does not include several drivers and software modules being packaged as add-ons such as Streams and NFS (Network File System). Drivers in the base system are installable drivers that have been delivered in the Base System Package of the Reliant UNIX 5.43 Foundation Set, rather than separate DSPs. They are just like other DSP's except that there are no *Install* or *Remove* scripts for the base drivers.

The *pkginfo* and *pkgrm* commands will not show these base drivers, not only to reduce clutter in those menus but also since it would be unreasonable to remove the base drivers. Although base drivers cannot be removed, they can be replaced with new drivers by installing an update driver software package (UDSP).

5.11.13 Update driver software package

This package is specifically designed to replace base system drivers. A UDSP must contain the following files:

- Those modules being replaced. Through special options of the ID commands used to install drivers, the old base driver's modules can be overlaid, removed, or supplemented. Those driver modules that are not changed do not have to be redelivered.
- The *postinstall* module. This module follows the same rules as for other driver packages except that it calls *idinstall* with the *-u* option.
- The *preremove* module. This module must print the message "Can not remove base driver" and return with an exit code of 1.

This scheme allows the user to install an UDSP just as any other ID package. When the user later uses the *pkginfo* command, the updated driver will be listed as "Device_Name Driver Update Package". The *pkgrm* menu will display the same entry, but if the user tries to select the updated driver, the *preremove* script defined above will abort the removal. If a subsequent update to that same driver is ever developed, the requirements for the UDSP are exactly the same as those itemized above for the first update. The second update will simply be loaded on top of the first. The *Name* file and the *Remove* file should remain the same in the second update package. This will cause the *pkgrm* and *pkginfo* command results to also remain the same.

It should be kept in mind that this update scenario is only for use with base drivers. If an add-on driver ever has an update, it is expected that the whole package previously installed will be removed, and the new release then re-installed.

5.11.14 Installation/removal summary

The ID commands and the DSP's modules defined above will be used together to rebuild and execute a new Reliant UNIX Operating System kernel. The step-by-step procedure to install, reconfigure, and execute a new kernel is as follows:

1. The User executes *pkgadd*.

   pkgadd loads files specified by *prototype* and executes *postinstall* script.
2. *postinstall* script will
   - Optionally prompt the user to determine hardware (IOA or IVN) strappings. This may include calling *idcheck* to test the usability of the IVN or IOA.
   - Execute */etc/conf/bin/idinstall* with the *-a* option. This command will
     - Move the DSP components to target directories
     - Update the file */etc/conf/cf.d/mdevice*
   - Execute */etc/conf/bin/idbuild*. This command will
     - Execute */etc/conf/bin/idconfig*
     - Execute */etc/conf/bin/idmkunix*
   - Install any user commands, menus, or files.
3. Upon finishing installation, a message to shutdown the system will be displayed.
4. After the user reboots:

   The *init* program is the first user-level program executed after reboot; */sbin/vc2* or */etc/vc2* will execute */etc/conf/bin/idmkenv*. This command tests to determine if this is the first boot of a new kernel. If so, the command will
   - Link */etc/conf/rc.d/** to */etc/idrc.d*
   - Link */etc/conf/sd.d/** to */etc/idsd.d*
   - Execute */etc/conf/bin/idmkinit*
   - Execute */etc/conf/bin/idmknod*
   - Continue *init* state 2 initialization

   The system boot will then continue normally.

The process of removing a DSP is very similar to this scenario with the exceptions that in step 1 the user invokes *pkgrm*, and in step 2, the *preremove* script will be deleting commands and files, and the *idinstall* command will be called with the *-d* option to delete the DSP. See [8] for a detailed description of this process.

5.11.15 Tunable system parameters

As mentioned earlier, there are two files which contain kernel tunable parameters: */etc/conf/cf.d/mtune* and */etc/conf/cf.d/stune*. These files can have a profound effect on system performance, and occasionally an add-on device driver or kernel software module may require you to modify an existing parameter or define a new tunable parameter that is accessible by other add-on drivers.

[7] and [9] provide manual pages for *mtune(4)* and *stune(4)*. As these pages show, the *mtune* file defines a default value along with a minimum and maximum value for each kernel parameter. An add-on package should never modify a predefined system parameter in the *mtune* file.

5.11.16 Modifying an existing kernel parameter

The *stune* file is used to modify a system-tunable parameter from its default value in the *mtune* file. Not every system-tunable parameter is contained in the *stune* file; only those that are to be set to a value other than the system default need be entered there. Although the base Reliant UNIX 5.43 defines only a few values in *stune*, other add-on packages may have added additional entries into *stune*. Therefore, if the driver package you are building requires modifying a parameter value, the *idtune* command should be used. See [7] for the manual page that describes *idtune(1M)*. This command will take individual system parameters, verify that the new value is within the upper and lower bounds specified in *mtune*, search the *stune* file, and modify an existing value if already there or add the parameter to *stune* if not defined.

The value selected must always be within the minimum and maximum values in the *mtune* file.

5.11.17 Defining a new kernel parameter

If the DSP you are developing is part of a group of kernel software components, there may be a need to define configurable parameters that other packages can reference. If this is the case, the *Install* script can append new tunable parameters to */etc/conf/cf.d/mtune* by defining lines in the format shown in the *mtune(4)* manual page in [9]. The DSP *Remove* script must remove these entries if the user chooses to remove the package. When modifying *mtune*, be careful that you do not modify or delete other values.

5.11.18 Reconfiguring the kernel to enable new parameters

After the *stune* and/or *mtune* files are modified, the system must be reconfigured using the *idbuild* command. If you are modifying the parameter as part of adding your DSP and your *Install* script already invokes *idbuild*, then, of course, no additional build is required.

5.11.19 Device driver development methodology

We have covered many of the kernel architectural and driver design details you need to know to write a Reliant UNIX device driver. Let's now talk about how you actually write the code and compile and package a driver. To accomplish these procedures, you must install the C Programming Language Utilities (CPLU) delivered in the C Software Development Set of the Reliant UNIX 5.43 Software Development Set.

As with any C program, you must compile, link edit, and execute the driver. Since the driver is part of the Kernel, it must be link edited together with the Kernel and the rest of the device drivers. The following can be used to create a driver object module suitable for the ID:

cc -c Driver.c

You can call the driver source by any name you wish as long as the object module is renamed *Driver.o* for later installation. If your driver is composed of several driver source files, they must each be compiled as above, then combined using *ld -r*. The resultant object module must be renamed *Driver.o*.

The ID requires that the driver object file be packaged on an installable floppy diskette along with the other modules described earlier. While you are initially developing and debugging the driver, it is not necessary to keep writing floppy diskettes and re-installing everything each time you make a driver modification. The

following section presents a methodology for driver development, debugging, and testing without the use of floppy installation packages.

5.11.20 Driver development procedures

Many of the steps that follow require you to modify files and directories owned by root. You must therefore be logged in as *root* or the system administrator to develop and debug device drivers. Throughout this section, the trace driver provided in Section "The trace driver", is used as a model.

1. Establish a device "Internal Name." This can be up to eight characters long and must start with a letter, but it can have digits or underscores after the first letter.

   It is the name that the ID uses to identify the device. For the trace driver, the name is "trace." From now on let's call this *DEV_NAME*. For the Reliant UNIX 5.43 ID implementation, the following name definitions based on the internal name are required:

   - Column 1 of the Master file. This must be *DEV_NAME*.
   - Column 4 of the Master file. This is the driver entry point (function) prefix. It is also called the "handler" field. It can be up to 6 characters. It is desirable to make this identical to column 1 if *DEV_NAME* is 4 characters or less. For the trace driver this prefix is "tr."
   - Column 1 of the System file. This must be *DEV_NAME*.
   - "Special file" names listed in the Node module. These should be *DEV_NAME0*, *DEV_NAME1*, etc., unless other issues, like user perception of the node name, are important. Any numbering for subdevices should match the minor device of that node. The trace driver package uses *trace0* which causes ID to generate */dev/trace0* on the first boot of the new kernel.
   - Function names inside your driver. The function names must use the device prefix defined above. The trace driver uses *tropen()*, *trclose()*, *trread()*, etc.
   - External variables and internal functions used inside the driver. These should use the prefix defined above or prefix followed by an underline. The trace driver uses "tr_".

2. Manually create a System entry. Go to the directory */etc/conf/sdevice.d*, and create a file of name *DEV_NAME* containing the System information. The trace driver uses the following:

□trace□□Y□□□1□□□0□□□0□□□0□□□0□□□0□□□0□□0

1. Manually create an *mdevice* entry. Since the ID assigns block and/or character major numbers when the package is installed, your Master file is required to have zeros in columns 5 and 6 , or 1-x, if multiple majors are required to request 'x' major numbers. Although you could manually edit */etc/conf/cf.d/mdevice* and assign block and character major numbers, the best approach is to put a file called *Master* in your local directory (say */tmp*) and execute the command:

   /etc/conf/bin/idinstall□-a□-m□-k□DEV_NAME

   This says add (*-a*) a Master entry (*-m*). Watch out! The *Master* file in your local directory will be removed by the *idinstall* command unless you use the *-k* option. The trace driver uses the following:

   trace□□□ocri□□□ioc□□□□tr□□□□0□□□□0□□□□1□□□□1□□□-1

   Once *idinstall* adds the Master entry, examine */etc/conf/cf.d/mdevice* and note the block and/or character major number.

2. Create a file in */etc/conf/node.d* to tell the ID to create device special files on the next system boot. The file should be named *DEV_NAME* and conform to the Node module format. For the trace driver the Node module is as follows:

   trace□□□trace□□0□□□c□□□0

3. Create */etc/conf/init.d*, */etc/conf/rc.d*, and */etc/conf/sd.d* entries if appropriate. This step can probably wait until debugging has proceeded.
4. Create a directory called */etc/conf/pack.d/DEV_NAME*. Put *Driver.o* and *Space.c* there (if you need them).
5. At this point, it would be a good idea to make a copy of your current Reliant UNIX Operating System kernel. Execute the following:

cp /stand/unix /stand/unix.bak

6. Manually execute the */etc/conf/bin/idbuild* shell script. This will run a configuration program and try to link edit your new driver into the Kernel. You will get an initial message followed either by a "Reliant UNIX System has been rebuilt" message or by error messages from the configuration program or link editor.

   If you get errors, correct them and repeat the above step. If the Kernel was built correctly a new Reliant UNIX System image will have been created in the */etc/conf/cf.d* directory. You can now shut the system down and reboot. Running */etc/shutdown* will cause the system to enter *init* state 0, 5 or 6 and the new kernel in */etc/conf/cf.d* will automatically be linked to */stand/unix*. On the next boot, if you specify */unix* on the boot: prompt, the new kernel will execute, and upon entering *init* state 2, the new device nodes, *inittab* entries, etc., will be installed.
7. When the system comes up, test your driver.

5.11.21 Emergency recovery

There is a possibility that the kernel will fail to boot if your driver contains a serious bug. This can be due to a *panic()* call that you put in your driver (see the next section) or some other system problem. If this happens, you should reset your system and boot your original kernel that you hopefully saved as recommended above. To do this, reset your machine, and when you see the *Booting Reliant UNIX System ...* message (machine-dependent), quickly strike the keyboard space bar to interrupt the default boot. When the boot prompt appears, type */unix.bak* or whatever you named your old kernel. If you did not save a copy of your kernel or some other disaster occurred, you can recover the system using the following emergency procedures to put a bootable */unix* image back on the hard disk:

1. Insert Floppy Diskette #1 of the Base System Software Set and push the RESET button on the front panel, or power the system down and then back up again.
2. Insert second diskette when prompted.

   When the prompt *Please press <RETURN> when ready to install the Reliant UNIX System* appears, press <DEL> to exit the installation program.

   You are now executing a floppy-bootable Reliant UNIX Operating System kernel. This is not a standard way to run the system. It should be used for emergency procedures only.
3. Execute the following commands:

   ```
   /etc/fs/bfs/fsck -y /dev/dsk/c0d0s10    # check the hard disk #
   /etc/fs/bfs/mount /dev/dsk/c0d0s10 /mnt # mount the hard disk #
   cp /stand/unix /mnt/unix                # copy a hard disk kernel #
   umount /mnt                             # unmount the hard disk #
   ```
4. Remove the floppy diskette.
5. Press the RESET Button or power down and then back up again.

   The system should now boot normally with a standard foundation Kernel. Your new driver and any other drivers you had installed on your system will not be included in */unix* even though they may appear in the *pkginfo* output. This can be corrected by removing your driver and executing

   /etc/conf/bin/idbuild.

   If that fails, the packages should be removed and re-installed.

This procedure can also be useful if other system files are damaged inadvertently while debugging your driver. There are several reasons your system may fail to boot properly or not let you log in after it has booted. For example, a corrupted password or *inittab* file could prevent console logins.

Since Floppy Diskette #1 of the *Base System Software Set* software contains a default */etc/passwd*, */etc/init*, */etc/inittab*, and other critical files, you can copy the default file from the floppy diskette to the root file system of the hard disk using the procedures above. Obviously, user logins you have added to */etc/passwd* or other system changes you have made since installing the original base system will be lost when you overwrite the corrupted file with the floppy diskette default file.

5.12 Driver debugging

5.12.1 Kernel print statements

There are, of course, limitations in debugging and testing device drivers. In the absence of a Kernel debugging tool, print statements inside the driver are the primary method used. *cmn_err()* calls made inside the Kernel will appear on the Reliant UNIX 5.43 monitor (*/dev/console*). Note that this *cmn_err()* is not the same as the *printf* in [9]. It has identical syntax to *printf(3)*, but it only supports print options byte, hexadecimal, character, decimal, unsigned decimal, octal, hexadecimal and string (option variables b, c, d, u, o, x, and s). See [16] for more information on *cmn_err()*.

Since the print statements are written by the Kernel, there is no way to redirect the output to a file or to remote terminal. Using print statements also modifies the timing of driver code execution, which may change the behavior of problems you are investigating.

Print statements in the driver can be made more efficient by using an *ioctl* to set one or more levels of debugging output. This way you can write a simple user program to turn the print output on or off as needed.

Sometimes kernel print statements scroll by too quickly to read. There is a limited kernel buffer called *putbuf* that records all kernel *printf*s. There are several ways to retrieve this data later :

Use the *crash* command. Try the following command after executing */etc/crash*:

od -a putbuf 2000

You can examine the *crash(1M)* manual page in [7] for more information.

Use the built-in kernel console monitor */dev/osm*. Since the base system does not have preconfigured */dev/osm* device nodes, you should make one:

Create a file named */etc/conf/node.d/osm* that contains the following:

osm osm0 c 0

Execute the */etc/conf/bin/idmknod* command.

Use *cat* or *tail* to examine */dev/osm0*.

cmn_err() has an option of putting the character data only in *putbuf* and not having the data appear on the console at all. This is done by preceding the text string with a "!". For example:

cmn_err(CE_NOTE,"!this driver print statement will only go into putbuf, not onto the screen.");

5.12.2 The trace driver

Another useful way to observe driver behavior is by using a trace driver. Such a driver can be called by your driver to log data. A user program can then be written to read the trace driver either in real-time or as a postmortem analysis. The main Section "The trace driver", provides the source code for such a driver which logs data presented to it by *trsave()* calls made from other drivers. The trace driver uses *clists* to save these traces. Although this driver isn't delivered with the base Reliant UNIX 5.43, you can compile and link edit the driver into your system from the source code presented in Section "The trace driver". Not only will */dev/trace0* be useful for your debugging, but it may help you better understand how the ID works before you actually write your driver.

5.12.3 System panics

If the programmer expects that the driver could enter a state that is illegal, the driver can halt the system by using the *cmn_err()* function, with a panic flag set. For example, if the driver expects one of three specific cases in a switch statement, the driver can add a fourth default case that calls the *cmn_err()* function. The system will dump an image of memory for later analysis. If the error is recoverable, the driver should not panic system. An example of panicking using *cmn_err()* is

cmn_err(CE_PANIC,"Your system has panic'ed, DEV_NAME error!");

5.12.4 Taking a system dump

This subsection is machine-specific.

In the event a *panic* occurs, there may be some value in examining the dump produced by the system. Since Reliant UNIX 5.43 uses the same physical hard disk slice for both "swap" and "dump," it is important that you

do not reboot to the multi-user state before examining the dump. If the system reaches multi-user state, the dump may be overwritten by system paging. To examine the dump you must save the dump image. Since the system detects an improper shutdown, you will receive a message as follows on the next reboot:

There□may□be□a□system□dump□memory□image□on□the□swap□device.
Do□you□want□to□save□it?□(y/n)>

Answer *y*. When given a selection list of what media to use for the dump say *q* for quad density 1.2 Megabyte floppy disks. (You will need a number of blank formatted floppy disks). Follow the instructions concerning floppy insertion and removal. When the image is written to a floppy disk, you will see a message reporting that */etc/ldsysdump* can be used to copy the dump off the disk. First, however, you must let the reboot continue its checking and remounting of the file systems.

When you see the *Console Login:* prompt, log in and execute */etc/ldsysdump* to take the dump off the floppy diskettes.

1. First do a *df* to determine a file system that has at least 8000 free disk blocks.
2. Execute "*ulimit 8000*" .
3. Execute *ldsysdump file*, where *file* is a file name to hold the dump image. It should be in the file system with ample room as found in step 1.
4. Follow the instructions and specify "q" for quad density disks. Insert the floppy disks as directed.

Finally, you can use the *crash* command to examine the dump as follows:

crash -d file

Consult the *crash(1M)* manual page in [7] for information on how to use *crash* to examine the Reliant UNIX Operating System kernel and user process status at the time of the panic. One useful piece of information might be to retrieve the panic printout and any other kernel messages that have made their way into *putbuf*. Use the *crash* command *od -a putbuf count* where *count* is the length of the *putbuf* data you wish to examine.

Note that the procedures to examine a memory dump only apply to Reliant UNIX Systems that have completed the dump sequence, usually in response to a *panic*. The prompt that you may see after an improper shutdown only indicates that the system was not properly brought down and a dump "may" exist. If the system is inadvertently powered down or reset, or if your device driver causes the kernel to hang or go berserk without ever executing a *panic*, no dump will have been taken, and the above procedures will yield a large memory image that *crash* will not be able to interpret. Remember, the following message applies only when you have properly detected an error and executed the *panic* function inside your driver or when your driver has caused a system error detected by the kernel or some other driver causing it to panic:

There□may□be□a□system□dump□memory□image□on□the□swap□device.
Do□you□want□to□save□it?□(y/n)>

At this point, it might be well to repeat the advice stated in the introduction:

Writing a device driver carries a heavy responsibility. As part of the Reliant UNIX Operating System kernel it is assumed to always take the correct action. Few limits are placed on the driver by the other parts

5.12.5 The postinstall script

When writing your script, keep the following rules in mind:

1. Use *idcheck* to determine whether your package is already installed and to verify the usability of IVNs/IOAs your driver and controller board use.
2. Call *idinstall* and exit appropriately on errors. Use the *echo* and/or *message* commands to tell the user what failed.
3. Call *idbuild* and check the return code. If non-zero, call *idinstall* to remove your package. If the driver fails kernel reconfiguration, don't leave it partially installed.

5.12.6 The preremove script

Although there are few reasons a remove operation will fail, the script should still remove the ID components and reconfigure the system first, then remove the user files. Check the return codes from the ID commands

and report to the user accordingly.

5.12.7 Writing the floppy diskette

Except for the requirement that the Size file be the first file on the floppy diskette, the ordering of your files is not important. The easiest way to do this is

1. put a formatted 360K or 1.2M diskette in the floppy drive
2. do a *cd* to where your files are
3. touch Size file
4. ls -t | cpio -ocB >/dev/rdsk/f0q15d (f0d9d for 360K floppies)

This procedure works only if all your files are in one directory with no subdirectories. If you use subdirectories, you will have to supply an explicit list of file names to *cpio* or use the *find* command in a way that puts the *Size* file first.

5.12.8 How to document your driver installation

This section is intended to give you some precautionary advice to pass on to your users. If you are developing a DSP that will be installed by users who may not be familiar with the implications of system reconfiguration, some words of caution may be worthwhile:

- Although experience has shown little difficulty installing and removing a variety of device drivers, there is the potential that a user may have difficulty booting the system. The cause of this would primarily be due to some fault in the added driver. If this occurred, the user would have to reload the Base System Set software, thus losing all user files. It may therefore be advisable to instruct users to back up user files before attempting an installation.
- Since a reconfiguration ends with a system reboot, it would not be advisable for other users to be logged on to the system through a remote terminal.
- The user should not hit <DEL> or <RESET>, power down the system, or in any way try to interrupt an installation. Although interruption protection is built into the ID scheme, total protection of a reboot during an installation can never be completely foolproof.
- Advise your users to run *df* and determine the free disk space before even trying an installation. Advise them of the number of free blocks needed to install the package.
- Advise the user not to have any background processes running that will
  1. be adversely affected by a system reboot
  2. consume free disk space while a reconfiguration is under way

  For example, running *uucp* during an installation should be avoided.

5.13 Device driver examples

5.13.1 The trace driver

The trace driver is a pseudo-device that allows the Reliant UNIX operating system kernel or other device drivers to report debugging information without the use of console *printf*s. The basic mechanism allows calls to the trace driver via *trsave()* to store short bursts of trace data in system character buffers (*clists*). These data items are retrieved from the *clists* and are reported to a user process by reading */dev/trace0*. This driver uses some additional calls common to other drivers, specifically, *sleep()*, *wakeup()*, and the *clist* handling routines.

In addition to providing the driver source code, other files needed to actually compile and use the trace device are provided:

trace.c
: The driver source code

trace.h
: The driver header file

Space.c

The DSP's memory allocation file

trsav.c
: A user program to read the trace device and redirect output to a disk file

trfmt.c
: A user program to print the trace information

If you wish to key this source code into your system, you can make use of this trace driver to debug a driver that you are writing.

The following notes help explain some of the driver source (*trace.c*) code:

- Lines 1-121:

  Represent the inclusion of system header files and define the open, close, and ioctl functions. The code is self-explanatory.

- Lines 122-149:

  The trace driver *read()* routine. The driver blocks (waits) until data is available via the *sleep()* function call. The read will block until the Kernel or some other driver issues a call to *trsave()*. When a *trsave()* call is made, trace data is put into a *clist* and a *wakeup()* is issued. The read awakens and transfers the trace data to the user process executing the read and releases the *clist*. Note the use of the internal trace driver address as the sleep event (*&tr_p->tr_rcnt*).

  Since *trsave()* calls can be done at interrupt level by other drivers, and since the *trsave()* function and the *trread()* function both manipulate the queue of *clists*, the *read* function surrounds its manipulation of the *clist* structures with *spl* calls.

- Lines 150-179:

  Data from other drivers is put into *clists*. Note that *trsave()* accesses the system time counter (*lbolt*), which represents time in ticks (1/100th of a second on 386 systems) since the system was booted. This places a time stamp on the trace event.

```
1    /*  Copyright (c) 1987 AT&T                                                */
2    /*    All Rights Reserved                                                  */
3    /*                                                                         */
4    /*  Space.c file for 386unix trace driver.                                 */
5    /*                                                                         */
6    /*  The trace structure defined here provides storage on a                 */
7    /*  per-subdevice basis. That is, one trace structure will be              */
8    /*  allocated for each sub-device. The variable TR_UNITS                   */
9    /*  is a #define created by the idconfig program. It represents            */
10   /*  the number of trace subdevices for the trace driver. It is             */
11   /*  derived from field 3 of the "System" entry for the device.             */
12   /*                                                                         */
13   /*  To locate TR_UNITS, this file should include config.h. This            */
14   /*  header file is created by the reconfiguration process and              */
15   /*  resides in the local directory of the reconfiguration                  */
16   /*  process (note use of double quotes around config.h).                   */
17   /*                                                                         */
18
19   #include "sys/types.h"
20   #include "sys/tty.h"
21   #include "sys/trace.h"
22   #include "config.h"
23
24   struct trace tr_data[TR_UNITS];
25   int tr_cnt=TR_UNITS;
26   /*        Copyright (c) 1987 AT&T          */
27   /*          All Rights Reserved            */
28
29   /*       386unix Trace Driver
30    *
31    * The trace driver is a pseudo-device that allows
32    * the Reliant UNIX Kernel or other device drivers to report\ debugging
```

```
33□□□□*□information□without□the□use□of□console□printf's.
34□□□□*□The□basic□mechanism□used□is□that□calls□to□the□trace□driver
35□□□□*□(via□trsave())□will□store□short□bursts□of□trace□data□in□system
36□□□□*□character□buffers□(clists).□These□data□items□are□retrieved\□from□the
37□□□□*□clists□and□are□reported□to□a□user□process□by□reading□/dev/trace.
38□□□□*
39□□□□*/
40
41□□□#include□"sys/types.h"
42□□□#include□"sys/signal.h"
43□□□#include□"sys/errno.h"
44□□□#include□"sys/param.h"
45□□□#include□"sys/dir.h"
46□□□#include□"sys/user.h"
47□□□#include□"sys/page.h"
48□□□#include□"sys/systm.h"
49□□□#include□"sys/tty.h"
50□□□#include□"sys/sysmacros.h"
51□□□#include□"sys/trace.h"
52
53□□□#define□OPEN□□□□□□□□□□□□□□□□□□□□□□01
54□□□#define□TRSLEEP□□□□□□□□□□□□□□□□□□04
55□□□#define□TRQMAX□□□□□□□□□□□□□□□□□□1024
56□□□#define□NIL□□□□□□□□□□□□□□□□□□□□□0377
57□□□#define□TRPRI□□□□□□□□□□□□□□□□□□□□(PZERO□+□3)
58
59□□□extern□int□tr_cnt;
60□□□extern□struct□trace□tr_data[□\];
61
62□□□tropen(dev)
63□□□{
64□□□□□□□□□int□chan;
65□□□□□□□□□register□struct□trace□*tr_p;
66
67□□□□□□□□□chan=minor(dev);
68□□□□□□□□□if□(chan□>=□tr_cnt)□{
69□□□□□□□□□□□□□□□u.u_error□=□ENXIO;
70□□□□□□□□□□□□□□□return;
71□□□□□□□□□}
72□□□□□□□□□tr_p□=□&tr_data[chan];
73□□□□□□□□□if□(tr_p->tr_state&OPEN)□{
74□□□□□□□□□□□□□□□u.u_error□=□EACCES;
75□□□□□□□□□□□□□□□return;
76□□□□□□□□□}
77□□□□□□□□□tr_p->tr_chno□=□NIL;
78□□□□□□□□□tr_p->tr_state□|=□OPEN;
79□□□}
80
81□□□trioctl(dev,□cmd,□arg,□mode)
82□□□{
83□□□□□□□□□register□struct□trace□*tr_p;
84□□□□□□□□□int□chan;
85
86□□□□□□□□□chan=minor(dev);
87□□□□□□□□□tr_p□=□&tr_data[chan];
88□□□□□□□□□switch(cmd)□{
89□□□□□□□□□case□TRACRCO:
90□□□□□□□□□□□□□□□tr_p->tr_chbits□|=□(01<<(int)arg);
91□□□□□□□□□□□□□□□return;
92□□□□□□□□□case□TRAGETC:
93□□□□□□□□□□□□□□□arg□=□tr_p->tr_chbits;
94□□□□□□□□□□□□□□□return;
95□□□□□□□□□case□TRASETC:
96□□□□□□□□□□□□□□□tr_p->tr_chbits□|=□arg;
```

```
97□□□□□□□□□□□□□return;
98□□□□□□□□□case□TRACLRC:
99□□□□□□□□□□□□□tr_p->tr_chbits□&=□~(short)arg;
100□□□□□□□□□□□□return;
101□□□□□□□□default:
102□□□□□□□□□□□□u.u_error□=□EINVAL;
103□□□□□□□□□□□□return;
104□□□□□□□□}
105□□}
106
107□□trclose(dev)
108□□{
109□□□□□□□□register□struct□trace□*tr_p;
110□□□□□□□□int□chan;
111
112□□□□□□□□chan=minor(dev);
113□□□□□□□□tr_p□=□&tr_data[chan];
114□□□□□□□□tr_p->tr_chbits□=□0;
115□□□□□□□□tr_p->tr_ct□=□0;
116□□□□□□□□tr_p->tr_chno□=□0;
117□□□□□□□□tr_p->tr_rcnt□=□0;
118□□□□□□□□while□(getc(&tr_p->tr_outq)>=0);
119□□□□□□□□tr_p->tr_state□=□0;
120□□}
121
122□□trread(dev)
123□□{
124□□□□□□□□register□struct□trace□*tr_p;
125□□□□□□□□int□chan;
126
127□□□□□□□□chan=minor(dev);
128□□□□□□□□tr_p□=□&tr_data[chan];
129□□□□□□□□spl5();
130□□□□□□□□tr_p->tr_state□|=□TRSLEEP;
131□□□□□□□□while□(tr_p->tr_rcnt□==□0)
132□□□□□□□□□□□□sleep((caddr_t)&tr_p->tr_rcnt,□TRPRI);
133□□□□□□□□spl0();
134□□□□□□□□while□(u.u_count□&&□tr_p->tr_rcnt)□{
135□□□□□□□□□□□□if□(tr_p->tr_chno□==□NIL)□{
136□□□□□□□□□□□□□□□□tr_p->tr_chno□=□getc(&tr_p->tr_outq);
137□□□□□□□□□□□□□□□□tr_p->tr_ct□=□getc(&tr_p->tr_outq);
138□□□□□□□□□□□□}
139□□□□□□□□□□□□if□(u.u_count□<□(tr_p->tr_ct□+□2))
140□□□□□□□□□□□□□□□□return;
141□□□□□□□□□□□□passc(tr_p->tr_chno);
142□□□□□□□□□□□□passc(tr_p->tr_ct);
143□□□□□□□□□□□□while□(tr_p->tr_ct𝑓𝑓)
144□□□□□□□□□□□□□□□□passc(getc(&tr_p->tr_outq));
145□□□□□□□□□□□□tr_p->tr_chno□=□NIL;
146□□□□□□□□□□□□tr_p->tr_rcnt𝑓𝑓;
147□□□□□□□□}
148□□}
149
150□□trsave(dev,□chno,□buf,□ct)
151□□int□dev,□chno,□ct;
152□□char□*buf;
153□□{
154
155□□□□□□□□register□struct□trace□*tr_p;
156□□□□□□□□register□int□n;
157□□□□□□□□register□char□*cpt;
158
159□□□□□□□□if□(dev□>=□tr_cnt)
160□□□□□□□□□□□□□□return;
```

```
      tr_p = &tr_data[dev];
      ct &= 0377;
      if ((tr_p->tr_chbits&(1<<chno)) == 0)
          return;
      if ((tr_p->tr_outq.c_cc + ct + 2 + sizeof(lbolt)) >TRQMAX)
          return;
      putc(chno, &tr_p->tr_outq);
      putc(ct + sizeof(lbolt), &tr_p->tr_outq);
      cpt = (char *)&lbolt;
      for (n = 0; n < sizeof(lbolt); ++n)
          putc(*cpt++, &tr_p->tr_outq);
      for (n=0;n<ct;n++)
          putc(buf[n], &tr_p->tr_outq);
      tr_p->tr_rcnt++;
      if (tr_p->tr_state&TRSLEEP) {
          tr_p->tr_state &= ~TRSLEEP;
          wakeup((caddr_t)&tr_p->tr_rcnt);
      }
}
/*  Copyright (c) 1985 AT&T                                              */
/*    All Rights Reserved                                                */
/*                                                                       */
/*  trsav - save 386unix event traces                                    */
/*                                                                       */
/*   usage: trsav mask device                                            */
/*                                                                       */
/*   Trsav opens the minor device of the trace driver specified  */
/*   by "device," enables the channels specified by "mask"\ (octal),*/
/*   and then reads event records and writes them to its standard*/
/*   output (unformatted) until killed.  Bit 0 of mask enables   */
/*   channel zero, bit 1 channel one, etc.                               */
/*                                                                       */
/*   For example, to enable saving of trace channel 0 from minor */
/*   device 0 of the trace driver and save the output in a file\ in  */
/*   /tmp, the following command can be used:                            */
/*                                                                       */
/*     trsav 1 /dev/trace0 > /tmp/temp.file &                            */

#include <stdio.h>
#include "sys/types.h"
#include "sys/tty.h"
#include "sys/trace.h"

char ev[512];
main(argc, argv)
char *argv[ \];
{
      int fd, n, k, seqno, chbits;
      if (argc != 3) {
          fprintf(stderr, "Incorrect number of arguments\n");
          fprintf(stderr, "Usage: trsav mask device\n");
          exit(1);
      }
      if ((fd = open(argv[2], 2)) < 0) {
          perror("trsav open:");
          exit(2);
      }
      setbuf(stdout, NULL);
      sscanf(argv[1], "%6o", &chbits);
      if ((k = ioctl(fd, TRASETC, chbits)) < 0) {
          perror("trsav ioctl:");
          exit(3);
      }
      seqno = 1;
```

```
225        for (;;) {
226            if ((n = read(fd, ev, 512)) < 0) {
227                perror("trsav read:");
228                exit(4);
229            }
230            if (write(1, ev, n) < 0) {
231                perror("trsav write:");
232                exit(5);
233            }
234        }
235    }
236    /*  Copyright (c) 1985 AT&T                                                    */
237    /*      All Rights Reserved                                                    */
238    /*                                                                             */
239    /*  386unix trace.h driver header file.                                        */
240    /*                                                                             */
241    /*  When the IOCTL defines are provided in a .h file,                          */
242    /*  both the driver and any user programs that need the IOCTL                  */
243    /*  values can work from the same set of #defines.                             */
244    /*                                                                             */
245    /*                                                                             */
246
247    /* IOCTL defines */
248    #define TRAC                ('T'<<8)
249    #define TRACMD              (TRAC|8)
250    #define TRAERRS             (TRAC|9)
251    #define TRARPT              (TRAC|10)
252    #define TRASDEV             (TRAC|11)
253    #define TRAATTACH           (TRAC|11)
254    #define TRADETACH           (TRAC|31)
255    #define TRAERRSET           (TRAC|32)
256    #define TRAERRGET           (TRAC|34)
257    #define TRAOPTS             (TRAC|33)
258    #define TRAPCDOPTS          (TRAC|35)
259    #define TRACRCO             (TRAC|16)
260    #define TRAGETC             (TRAC|17)
261    #define TRASETC             (TRAC|18)
262    #define TRACLRC             (TRAC|19)
263    #define TRASTAT             (TRAC|36)
264
265    /*
266    *  Per trace structure
267    */
268    struct trace {
269        struct clist tr_outq;
270        short tr_state;
271        short tr_chbits;
272        short tr_rcnt;
273        unsigned char tr_chno;
274        char tr_ct;
275    };
276    /*    Copyright (c) 1987 AT&T                                                  */
277    /*        All Rights Reserved                                                  */
278    /*                                                                             */
279    /*                                                                             */
280    /*    trfmt - print 386unix event traces                                       */
281    /*                                                                             */
282    /*    Trfmt reads its standard input, which it assumes was                     */
283    /*    generated by trsav, and prints it (formatted) to                         */
284    /*    standard output until killed.  Trfmt can read a file                     */
285    /*    written by trsav or except pipe output as follows:                       */
286    /*                                                                             */
287
288    /*                trfmt < /tmp/temp.file                                       */
```

```
  /*                              or                                                                                        */
  /*                              trsav mask device | trfmt                                                 */
  /*                                                                                                                          */
  /*   This version will format and print predefined lines of text */
  /*   for only a few types of typical driver traces: O is "open," */
  /*   C is "close," etc. If you wish to use other trace points in */
  /*   your driver, define your own trace identifiers and add them */
  /*   to the case statement below.                                                             */
  /*                                                                                                                          */
  /*                                                                                                                          */

  #include <stdio.h>

  #define  MASK16           0177777
  #define  SSTOL(x, y)      ((((long)x)<<16)|(((long)y)&MASK16))

  struct event {
        unsigned short  lbolt1;
        unsigned short  lbolt2;
        unsigned short  seq;
        unsigned char   typ;
        unsigned char   dev;
        unsigned short  wd1;
        unsigned short  wd2;
  } ev;
  main(argc, argv)
  char *argv[ \];
  {
        extern int optind;
        int x, fd, k, n, seqno, con;
        char *type;
        long time1;
        char xxx;

  setbuf(stdout,NULL);
        seqno = 1;
        for (;;) {
            x = getchar();
            n = getchar();
            if ((k=fread((char *)&ev, sizeof(xxx), n, stdin))< 0){
                perror();
                exit(3);
            }
            if (k == 0) {
                clearerr(stdin);
                sleep(1);
                continue;
            }
            if (ev.seq != seqno)
                printf("**%d event records lost**\n",
                    ev.seq - seqno);
            seqno = ev.seq + 1;
            if (k == 12) {
                time1 = SSTOL(ev.lbolt2, ev.lbolt1);
                printf(" %10lu%6d", time1, ev.seq);
                switch((int)ev.typ){
                case 'W':
                    type = "write";
                    printf(" %-8s %2o%6o%6o", type, ev.dev,
                    ev.wd1, ev.wd2);
                    break;
                case 'R':
                    type = "read";
                    printf(" %-8s %2o%6o%6o", type, ev.dev,
```

```
353□□□□□□□□□□□□□□□□□□□□□ev.wd1,□ev.wd2);
354□□□□□□□□□□□□□□□□□□□□□break;
355□□□□□□□□□□□□□□□□□case□'O':
356□□□□□□□□□□□□□□□□□□□□□type□=□"open";
357□□□□□□□□□□□□□□□□□□□□□printf("□%-8s□%2o%6o%6o",□type,□ev.dev,
358□□□□□□□□□□□□□□□□□□□□□ev.wd1,□ev.wd2);
359□□□□□□□□□□□□□□□□□□□□□break;
360□□□□□□□□□□□□□□□□□case□'C':
361□□□□□□□□□□□□□□□□□□□□□type□=□"close";
362□□□□□□□□□□□□□□□□□□□□□printf("□%-8s□%2o%6o%6o",□type,□ev.dev,
363□□□□□□□□□□□□□□□□□□□□□ev.wd1,□ev.wd2);
364□□□□□□□□□□□□□□□□□□□□□break;
365□□□□□□□□□□□□□□□□□case□'I':
366□□□□□□□□□□□□□□□□□□□□□type□=□"ioctl";
367□□□□□□□□□□□□□□□□□□□□□printf("□%-8s□%2o%7o%7o",type,ev.dev,
368□□□□□□□□□□□□□□□□□□□□□ev.wd1,□ev.wd2);
369□□□□□□□□□□□□□□□□□□□□□break;
370□□□□□□□□□□□□□□□□□/*
371□□□□□□□□□□□□□□□□□case□'?':
372□□□□□□□□□□□□□□□□□*□□Place□custom□driver□reports□here.
373□□□□□□□□□□□□□□□□□*□□Drivers□or□Kernel□functions□which□call
374□□□□□□□□□□□□□□□□□*□□trsave()□can□use□any□type□definitions
375□□□□□□□□□□□□□□□□□*□□and/or□print□formats□deemed□appropriate.
376□□□□□□□□□□□□□□□□□*/
377□□□□□□□□□□□□□□□□□default:
378□□□□□□□□□□□□□□□□□□□□□printf("□%-10c%2o%7o%6o",□ev.typ,
379□□□□□□□□□□□□□□□□□□□□□ev.dev,□ev.wd1,□ev.wd2);
380□□□□□□□□□□□□□□□□□}
381□□□□□□□□□□□□□□□□□printf("\n");
382□□□□□□□□□□□□□}
383□□□□□□□□}
384□□}
```

5.14 A sample driver software package

This section contains sample files needed to install a device driver that is part of an OA&M-style installable software package. The principal files are the *postinstall* and *preremove* files. These files contain shell scripts that are used to install and remove the device driver. The *prototype* file is used to install any commands or header files.

The driver package described in the section "The trace driver" is provided here as an example. Although the driver is a software driver (and hence will not contain hardware-related examples that are needed for hardware driver installation), most of the content of this driver package relates to any device driver.

The *postinstall* script presented here contains a large amount of diagnostic and recovery information such as checking if the driver is already installed and overwriting the old driver if the user confirms. All errors are redirected to a file in */tmp*. It is up to the *postinstall* script to deal with what errors the user should and shouldn't see on the screen.

Some items to note in the *postinstall* script:

- make liberal use of the *echo* and *message* commands to tell the user what is going on
- make sure you exit with the appropriate return value based on successful or non-successful installation

```
#□Sample□OA&M□package□driver□postinstall□script
#
#□Assumes□driver□object□file□and□related□ID□files□copied□into□/tmp/trace.
#□Will□only□allow□driver□to□be□installed□on□UNIX□SVR4.0□system
FAILURE=1□□□□□□□□#□fatal□error
SUCCESS=10□□□□□□□□#□success
TMP=/tmp/trace.err
ERROR1="□Errors□have□been□written□to□the□file□$TMP."
CONFDIF=${ROOT}/etc/conf
CONFBIN=${CONFDIF}/bin
PACK=${CONFDIF}/pack.d
```

```
NOTOOLS="ERROR:□The□Installable□Driver□feature□has□been□removed.
The□${NAME}□cannot□be□installed."
PARTINS="WARNING:□A□TRACE□Driver□has□been□partially□installed.□It□is□unknown
□□□□□□□□how□completely□it□is□installed.□You□may□continue□and□overlay□it□with
□□□□□□□□the□${NAME}."
BASE1="ERROR:□The□${NAME}□is□not□compatible□with□this□release□of□the□Reliant□UNIX
□□□□□□□V5.4x□operating□system□and□can□not□be□used□with□this□system."
rm□-f□$TMP□>□/dev/null□2>&1□□□□□□□□#□remove□any□existing□error□file
#□determine□that□ID/TP□tools□are□available
if
□□□□□□[□-x□${CONFBIN}/idcheck□-a□-x□${CONFBIN}/idbuild
□□□□□□□□-a□-x□${CONFBIN}/idinstall□]
then
□□□□□□□□:
else
□□□□□□□□message□${NOTOOLS}
□□□□□□□□exit□$FAILURE
fi
cd□/tmp/trace
#□verify□installation□on□Reliant□UNIX□V5.4x
OSVER=`uname□-v`
case□${OSVER}□in
□□□□□□□□5.4x□*)□;;
□□□□□□□□*)□message□${BASE1};
□□□□□□□□□□□exit□$FAILURE;;
esac
${CONFBIN}/idcheck□-p□trace□>□/dev/null□2>&1
RETTP=$?
□□**********************************************************************
□□##□If□RETTP□!=□0,□then□an□"trace"□driver□exists□on□the□system□##
□□**********************************************************************
WARN=""
if□[□$RETTP□!=□0□]
then
□□□□□□□□message□-c□${PARTINS}
□□□□□□□□if□[□"$?"□!=□"0"□]
□□□□□□□□then
□□□□□□□□□□□□□□□□□exit□${FAILURE}
□□□□□□□□fi
□□□□□□□□idinstall□-d□trace□□□□□□□□#□remove□current□copy
fi
${CONFBIN}/idinstall□-a□trace□2>>□$TMP
if□[□$?□!=□0□]
then
□□□□□□□□message□"The□installation□cannot□be□completed□due□to□an□error□in□the
□□□□□□□□□□□□□□□□□□driver□installation.□$ERROR1□Please□try□the□installation
□□□□□□□□□□□□□□□□□□again.□If□the□error□occurs□again,contact□your□Trace□Service
□□□□□□□□□□□□□□□□□□Representative."
□□□□□□□□exit□${FAILURE}
fi
${CONFBIN}/idinstallf□-f□trace
${CONFBIN}/idbuild□2>>□$TMP
if
□□□□□□□□[□"$?"□-ne□"0"□]
then
□□□□□□□□message□"The□installation□cannot□be□completed□due□to□an□error□in□the
□□□□□□□□□□□□□□□□□□kernel□reconfiguration.□$ERROR1□Please□try□the□installation
□□□□□□□□□□□□□□□□□□again.□If□the□error□occurs□again□,□contact□your□Trace□Service
□□□□□□□□□□□□□□□□□□Representative."
□□□□□□□□exit□${FAILURE}
fi
#□Needed□so□the□removef□works.
removef□$PKGINST□/tmp/trace/*□>/dev/null□2>&1
removef□-f□$PKGINST□>/dev/null□2>&1
```

```
rm□-f□$TMP□1>/dev/null□2>&1
exit□${SUCCESS}
------------------------------
#preremove□script□for□trace□driver.
FAILURE=1□□□□□□□□#□fatal□error
SUCCESS=10
CONFDIR=/etc/conf
CONFBIN=$CONFDIR/bin
NOTOOLS="ERROR:□The□Installable□Driver□feature□has□been□removed.
□□□□□□□□□□□□□□□□□The□${NAME}□cannot□be□removed."
TMP=/tmp/trace.err1
ERROR1="An□an□error□was□encountered□removing□the□${NAME}.□□The□file□${TMP}
□□□□□□□□contains□errors□reported□by□the□system."
ERROR2="The□kernel□rebuild□failed.□□However□all□software□dealing□with□the
□□□□□□□□${NAME}□has□been□removed.□The□${NAME}□will□still□appear□in□the□Show
□□□□□□□□Installed□Software□/Remove□Installed□Software□menus□because□the□kernel
□□□□□□□□still□has□the□trace□driver□in□it.□Please□correct□the□problem□and
□□□□□□□□remove□the□software□again.□The□file□$TMP□contains□error□reported□by
□□□□□□□□the□system."
rm□-f□$TMP□>□/dev/null□2>&1
if□[□-x□${CONFBIN}/idcheck□-a□-x□${CONFBIN}/idbuild
□□□□□-a□-x□${CONFBIN}/idinstall□]
then
□□□□□□□□:
else
□□□□□□□□message□${NOTOOLS}
□□□□□□□□exit□${FAILURE}
fi
${CONFBIN}/idcheck□-p□trace
RES="$?"
if
□□□□□□□□[□"${RES}"□-ne□"100"□-a□"${RES}"□-ne□"0"□]
then
□□□□□□□□${CONFBIN}/idinstall□-d□trace□□2>>□${TMP}
□□□□□□□□if□[□$?□!=□0□]
□□□□□□□□then
□□□□□□□□□□□□□□□□□IDERR=1
□□□□□□□□fi
□□□□□□□□REBUILD=1
fi
if□[□${IDERR}□!=□0□]
then
□□□□□□□□message□$ERROR1
□□□□□□□□exit□${FAILURE}
fi
RETVAL=0
if
□□□□□□□□[□"${REBUILD}"□=□"1"□]
then
□□□□□□□□#□rebuild□for□changes□to□take□effect
□□□□□□□□${CONFBIN}/idbuild□2>>□$TMP
□□□□□□□□if□[□$?□!=□0□]
□□□□□□□□then
□□□□□□□□□□□□□□□□□message□$ERROR2
□□□□□□□□□□□□□□□□□exit□${FAILURE}
□□□□□□□□else
□□□□□□□□□□□□□□□□□RETVAL=${SUCCESS}
□□□□□□□□fi
fi
rm□-f□$TMP□1>/dev/null□2>&1
exit□${RETVAL}
------------------------------------------------
The□pkginfo□file
#
```

```
#□sample□OA&M□pkginfo□file
#
PKG="trace"
CLASSES="none"
NAME="386unix□Trace□Device□Driver□Package"
RELEASE="4.0"
VERSION="1"
VENDOR="SNI"
CATEGORY=system
#□ARCH□is□set□to□i386□because□the□trace□driver□is□not□specific□to□a
particular□386□architecture□ARCH="i386"
#□The□following□allows□old□displaypkg□command□to□show□Trace□Driver□package□as
installed□PREDEPEND="trace"
-----------------------------------------------
The□prototype□file
#□sample□OA&M□package□prototype□file
#□PACKDIR□is□where□built□Driver.o□and□related□ID/TP□files□are□located.
#□We□use□/usr/src/pkg/trace□as□an□example
!PACKDIR=/usr/src/pkg/trace
#□the□following□files□should□be□in□the□same□directory□as□the□prototype□file
i□pkginfo
i□postinstall
i□preremove
!PKGINST=trace
!PKGSAV=/var/sadm/pkg/$PKGINST/save
#□class□"v"□files□are□volatile□-□-□allowed□to□be□removed□by□package
installation
!default□0544□bin□bin
d□none□/tmp/trace□□□□□□□□□□□□□□□775□□□□□□bin□□□□□□□□□bin
v□none□/tmp/trace/Driver.o=$PACKDIR/Driver.o□□□□□?□□□□□□□□□?□□□□□□□□□?
v□none□/tmp/trace/Master=$PACKDIR/Master□□□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
v□none□/tmp/trace/System=$PACKDIR/System□□□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
v□none□/tmp/trace/Node=$PACKDIR/Node□□?□□□□□□□□□□□?□□□□□□□□□?
v□none□/tmp/trace/Rc=$PACKDIR/Rc□□□□□□□?□□□□□□□□□□□?□□□□□□□□□?
v□none□/tmp/trace/Space.c=$PACKDIR/Space.c□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
#□□□□□□□□directories:□default□owner=root□group=sys□mode=0775
!default□0544□root□sys
d□none□□□□□□□□□/usr□□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
d□none□□□□□□□□□/usr/bin□□□□□□□□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
d□none□□□□□□□□□/usr/include□□□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
d□none□□□□□□□□□/usr/include/sys□□□□□□□□□□□□□□□□?□□□□□□□□□?□□□□□□□□□?
#□assume□"trace"□is□a□command□part□of□the□trace□driver□package
f□none□□□□□□□□□/usr/bin/trace=$PACKDIR/trace
#□□□□□□□□□header□files:□default□owner=bin□group=bin□mode=0444
!default□0444□bin□bin
f□none□□□□□□□□□/usr/include/sys/trace.h=$PACKDIR/trace.h
-------------------------------------------------------------------------
The□Master□file:
----------------
trace□□□ocri□□□□ioc□□□□□tr□□□□□□□0□□□□□□□□0□□□□□□□□1□□□□□□□□1□□□□□□□-1
-------------------------------------------------------------------------
The□System□file:
----------------
trace□□□Y□□□□□□□□1□□□□□□□□0□□□□□□□□0□□□□□□□□0□□□□□□□□0□□□□□□□□0□□□□□□□□0□□□□□□□□0
-------------------------------------------------------------------------
The□Node□file:
--------------
trace□□□trace0□□c□□□□□□□0
-------------------------------------------------------------------------
```

6 Reference pages

The manual pages which form this section are from the manual "Programmer's Guide: System Services and Application Packaging Tools" [2]. There are various other manual pages which are in some way related to the topics dealt with in this manual but have not been repeated here. For such manual pages you should refer to the appropriate "Reference Manual".

NAME

ckdate, errdate, helpdate, valdate - prompts for and validates a date

SYNOPSIS

ckdate
[-Q] [-W *width*] [-f *format*] [-d *default*] [-h *help*] [-e *error*]
[-p *prompt*] [-k *pid*[-s *signal*]]

errdate
[-W] [-e *error*] [-f *format*]

helpdate
[-W] [-h *help*] [-f *format*]

valdate
[-f *format*] *input*

DESCRIPTION

ckdate prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a date, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). The user response must match the defined format for a date.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The -W option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckdate* command. They are *errdate* (which formats and displays an error message), *helpdate* (which formats and displays a help message), and *valdate* (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *format* is defined in the *errdate* and *helpdate* modules, the messages will describe the expected format.

OPTIONS

-Q
Specifies that *quit* will not be allowed as a valid response.

-W *width*
Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-f *format*
Specifies the format against which the input will be verified. Possible formats and their definitions are:

%b=
abbreviated month name

%B=
full month name

%d=
day of month (01 - 31)

%D=
date as *%m/%d/%y* (the default format)
%e=
day of month (1 - 31; single digits are preceded by a blank)
%h=
abbreviated month name (*jan*, *feb*, *mar*)
%m=
month number (01 - 12)
%y=
year within century (e.g. 89)
%Y=
year as *CCYY* (e.g. 1989)

-d *default*
Defines the default value as *default*. The default does not have to meet the format criteria.

-h *help*
Defines the help messages as *help*.

-e *error*
Defines the error message as *error*.

-p *prompt*
Defines the prompt message as *prompt*.

-k *pid*
Specifies that process ID *pid* is to be sent a signal if the user choosesto abort.

-s *signal*
Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
Input to be verified against format criteria.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

4 = Garbled format argument

NOTES

The default prompt for *ckdate* is:

Enter the date [?,q]:

The default error message is:

ERROR - Please enter a date, using the following format: <format>.

The default help message is:

Please enter a date, using the following format: <format>.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valdate* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckgid, errgid, helpgid, valgid - prompts for and validates a group id

SYNOPSIS

ckgid
[-Q] [-W *width*] [-m] [-d *default*] [-h *help*] [-e *error*]□
[-p *prompt*] [-k *pid*[-s *signal*]]

errgid
[-W] [-e *error*]

helpgid
[-W] [-m] [-h *help*]

valgid
input

DESCRIPTION

ckgid prompts a user and validates the response. It defines, among other things, a prompt message whose response should be an existing group ID, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckgid* command. They are *errgid* (which formats and displays an error message), *helpgid* (which formats and displays a help message), and *valgid* (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt.

OPTIONS

-Q
Specifies that *quit* will not be allowed as a valid response.

-W *width*
Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-m
Displays a list of all groups when help is requested or when the user makes an error.

-d *default*
Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h *help*
Defines the help messages as *help*.

-e *error*
Defines the error message as *error*.

-p *prompt*
Defines the prompt message as *prompt*.

-k *pid*
Specifies that process ID *pid* is to be sent a signal if the user aborts with *q*.

-s *signal*
Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
Input to be verified against */etc/group*.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

NOTES

The default prompt for *ckgid* is:

Enter the name of an existing group [?,q]:

The default error message is:

ERROR - Please enter the name of an existing group.

If the *-m* option of *ckgid* is used, a list of valid groups is displayed here.

The default help message is:

Please enter an existing group name.

If the *-m* option of *ckgid* is used, a list of valid groups is displayed here.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valgid* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckint - display a prompt; verify and return an integer value

SYNOPSIS

ckint

[-Q] [-W *width*] [-b *base*] [-d *default*] [-h *help*] [-e *error*]□
[-p *prompt*] [-k *pid*[-s *signal*]]

errint

[-W] [-b *base*] [-e *error*]

helpint

[-W] [-b *base*] [-h *help*]

valint

[-b *base*] *input*

DESCRIPTION

ckint prompts a user, then validates the response. It defines, among other things, a prompt message whose response should be an integer, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckint* command. They are *errint* (which formats and displays an error message), *helpint* (which formats and displays a help message), and *valint* (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *base* is defined in the *errint* and *helpint* modules, the messages will include the expected base of the input.

OPTIONS

-Q

Specifies that *quit* will not be allowed as a valid response.

-W
: Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-b
: Defines the base for input. Must be 2 to 36, default is 10.

-d
: Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
: Defines the help messages as *help*.

-e
: Defines the error message as *error*.

-p
: Defines the prompt message as *prompt*.

-k
: Specifies that process ID *pid* is to be sent a signal if the user aborts with *q*.

-s
: Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
: Input to be verified against *base* criterion.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

NOTES

The default base 10 prompt for *ckint* is:

Enter an integer [?,q]:

The default base 10 error message is:

ERROR - Please enter an integer.

The default base 10 help message is:

Please enter an integer.

The messages are changed from *integer* to *base base integer* if the base is set to a number other than 10.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valint* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckitem - build a menu; prompt for and return a menu item

SYNOPSIS

ckitem
: [-Q] [-W *width*] [-uno] [-f *file*] [-l *label*] [[-i *invis*][, ...]]□
[-m *max*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*]□
[-k *pid* [-s *signal*]] [*choice*[...]]

erritem
: [-W] [-e *error*] [*choice* [...]]

helpint

[-W] [-h *help*] [*choice* [...]]

DESCRIPTION

ckitem builds a menu and prompts the user to choose one item from a menu of items. It then verifies the response. Options for this command define, among other things, a prompt message whose response will be a menu item, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

By default, the menu is formatted so that each item is preceded by a number and is printed in columns across the terminal. Column length is determined by the longest choice. Items are alphabetized.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Two visual tool modules are linked to the *ckitem* command. They are *erritem* (which formats and displays an error message) and *helpitem* (which formats and displays a help message). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *choice* is defined in these modules, the messages will describe the available menu choice (or choices).

OPTIONS

-Q
: Specifies that *quit* will not be allowed as a valid response.

-W
: Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-u
: Specifies that menu items should be displayed as an unnumbered list.

-n
: Specifies that menu items should not be displayed in alphabetical order.

-o
: Specifies that only one menu token will be returned.

-f
: Defines a file, *file*, which contains a list of menu items to be displayed. The format of this file is: *token<tab>description*. Lines beginning with a hash sign (#) are designated as comments and ignored.

-l
: Defines a label, *label*, to print above the menu.

-i
: Defines invisible menu choices (those which will not be printed in the menu). For example, "all" used as an invisible choice would mean it is a legal option but does not appear in the menu. Any number of invisible choices may be defined. Invisible choices should be made known to a user either in the prompt or in a help message.

-m
: Defines the maximum number of menu choices allowed.

-d
: Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
: Defines the help messages as *help*.

-e
: Defines the error message as *error*.

-p
: Defines the prompt message as *prompt*.

-k
: Specifies that the process ID *pid* is to be sent a signal if the user aborts with *q*.

-s
: Specifies that process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

choice
: Defines menu items. Items should be separated by white space or newline.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

4 = No choices from which to choose

SEE ALSO

allocmenu(3X), printmenu(3X), setinvis(3X), setitems(3X)

NOTES

The user may input the number of the menu item if choices are numbered or as much of the string required for a unique identification of the item. Long menus are paged with 10 items per page.

When menu entries are defined both in a file (by using the *-f* option) and also on the command line, they are usually combined alphabetically. However, if the *-n* option is used to suppress alphabetical ordering, then the entries defined in the file are shown first, followed by the options defined on the command line.

The default prompt for *ckitem* is:

Enter selection [?,??,q]:

One question mark will give a help message and then redisplay the prompt. Two question marks will give a help message and then redisplay the menu label, the menu and the prompt.

The default error message is:

ERROR - Does not match an available menu selection.

Enter one of the following:

- the number of the menu item you wish to select
- the token associated withe the menu item,
- partial string which uniquely identifies the token for the menu item
- ?? to reprint the menu

The default help message is:

Enter one of the following:

- the number of the menu item you wish to select
- the token associated with the menu item,
- partial string which uniquely identifies the token for the menu item
- ?? to reprint the menu

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*.

NAME

ckkeywd - prompts for and validates a keyword

SYNOPSIS

ckkeywd

[-Q] [-W *width*] [-d *default*] [-h *help*] [-e *error*] [-p *prompt*]□
[-k *pid* [-s *signal*]] [*keyword* [...]]

DESCRIPTION

ckkeywd prompts a user and validates the response. It defines, among other things, a prompt message whose response should be one of a list of keywords, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). The answer returned from this command must match one of the defined list of keywords.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

OPTIONS

-Q
: Specifies that quit will not be allowed as a valid response.

-W
: Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-d
: Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
: Defines the help messages as *help*.

-e
: Defines the error message as *error*.

-p
: Defines the prompt message as *prompt*.

-k
: Specifies that process ID *pid* is to be sent a signal if the user aborts with *quit*.

-s
: Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

keyword
: Defines the keyword, or list of keywords, against which the answer will be verified.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

4 = No keywords from which to choose

NOTES

The default prompt for ckkeywd is:

Enter selection [keyword,[...],?,q]:

The default error message is:

ERROR - Does not match any of the valid selections.

Please enter one of the following keywords:
keyword[,...]

The default help message is:

Please enter one of the following keywords:
keyword[,...]

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*.

NAME

ckpath - display a prompt; verify and return a pathname

SYNOPSIS

ckpath
[-Q] [-W *width*] [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] □
[-d *default*] [-h help] [-e error] [-p prompt] [-k pid[-s signal]]

errpath
[-W] [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] [-e *error*]

helppath
[-W] [-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] [-h *help*]

valpath
[-a|l] [-b|c|g|y] [-n|[o|z]] [-rtwx] *input*

DESCRIPTION

ckpath prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a pathname, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

The pathname must obey the criteria specified by the first group of options. If no criteria is defined, the pathname must be for a normal file that does not yet exist. If neither *-a* (absolute) or *-l* (relative) is given, then either is assumed to be valid.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckpath* command. They are *errpath* (which formats and displays an error message), *helppath* (which formats and displays a help message), and *valpath* (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

OPTIONS

-Q
Specifies that *quit* will not be allowed as a valid response.

-W
Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-a
Pathname must be an absolute path.

-l
Pathname must be a relative path.

-b
Pathname must be a block special file.

-c

Pathname must be a character special file.

-g
Pathname must be a regular file.

-y
Pathname must be a directory.

-n
Pathname must not exist (must be new).

-o
Pathname must exist (must be old).

-z
Pathname must have a length greater than 0 bytes.

-r
Pathname must be readable.

-t
Pathname must be creatable (touchable). Pathname will be created if it does not already exist.

-w
Pathname must be writable.

-x
Pathname must be executable.

-d
Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
Defines the help messages as *help*.

-e
Defines the error message as *error*.

-p
Defines the prompt message as *prompt*.

-k
Specifies that process ID *pid* is to be sent a signal if the user aborts with *quit*.

-s
Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
Input to be verified against validation options.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

4 = Mutually exclusive options

NOTES

The text of the default messages for *ckpath* depends upon the criteria options that have been used. An example default prompt for *ckpath* (using the *-a* option) is:

Enter a pathname [?,q]:

An example default error message (using the *-a* option) is:

ERROR - Invalid pathname entered.
A pathname is a filename, optionally preceded by parent directories.

An example default help message is:

A pathname is a filename, optionally preceded by parent directories.
The pathname you enter:

- must contain 1 to {NAME_MAX} characters
- must not contain a spaces or special characters

NAME_MAX is a system variable that is defined in *limits.h*.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valpath* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckrange - prompts for and validates an integer

SYNOPSIS

ckrange
[-Q] [-W *width*] [-l *lower*] [-u *upper*] [-b *base*] [-d *default*]
[-h *help*] [-e *error*] [-p *prompt*] [-k *pid*[-s *signal*]]

errange
[-W] [-l *lower*] [-u *upper*] [-e *error*]

helprange
[-W] [-l *lower*] [-u *upper*] [-h *help*]

valrange
[-l *lower*] [-u *upper*] [-b *base*] *input*

DESCRIPTION

ckrange prompts a user and validates the response. It defines, among other things, a prompt message whose response should be an integer in the range specified, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

This command also defines a range for valid input. If either the lower or upper limit is left undefined, then the range is bounded on only one end.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckrange* command. They are *errange* (which formats and displays an error message), *helprange* (which formats and displays a help message), and *valrange* (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

OPTIONS

-Q
Specifies that *quit* will not be allowed as a valid response.

-W
Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-l
Defines the lower limit of the range as *lower*. Default is the machine's largest negative integer or long.

-u
Defines the upper limit of the range as *upper*. Default is the machine's largest positive integer or long.

-b
: Defines the base for input. Must be 2 to 36, default is 10.

-d
: Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
: Defines the help messages as *help*.

-e
: Defines the error message as *error*.

-p
: Defines the prompt message as *prompt*.

-k
: Specifies that process ID *pid* is to be sent a signal if the user aborts with *quit*.

-s
: Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
: Input to be verified against upper and lower limits and base.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

NOTES

The default base 10 prompt for *ckrange* is:

Enter an integer between lower_bound and upper_bound [q,?]:

The default base 10 error message is:

ERROR - Please enter an integer between lower_bound and upper_bound.

The default base 10 help message is:

Please enter an integer between lower_bound and upper_bound.

The messages are changed from *integer* to *base base integer* if the base is set to a number other than 10.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valrange* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckstr - display a prompt; verify and return a string answer

SYNOPSIS

ckstr
: [-Q] [-W *width*] [[-r *regexp*][...]] [-l *length*] [-d *default*]□
[-h *help*] [-e *error*] [-p *prompt*] [-k *pid*[-s *signal*]]

errstr
: [-W] [-e *error*]

helpstr
: [-W] [-h *help*]

valstr
: *input*

DESCRIPTION

ckstr prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a string, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

The answer returned from this command must match the defined regular expression and be no longer than the length specified. If no regular expression is given, valid input must be a string with a length less than or equal to the length defined with no internal, leading or trailing white space. If no length is defined, the length is not checked. Either a regular expression or a length must be given with the command.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckstr* command. They are *errstr* (which formats and displays an error message), *helpstr* (which formats and displays a help message), and *valstr* (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

OPTIONS

-Q
: Specifies that *quit* will not be allowed as a valid response.

-W
: Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-r
: Specifies a regular expression, *regexp*, against which the input should be validated. May include white space. If multiple expressions are defined, the answer must match only one of them.

-l
: Specifies the maximum length of the input.

-d
: Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
: Defines the help messages as *help*.

-e
: Defines the error message as *error*.

-p
: Defines the prompt message as *prompt*.

-k
: Specifies that process ID *pid* is to be sent a signal if the user aborts with *quit*.

-s
: Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
: Input to be verified against format length and/or regular expression criteria.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

NOTES

The default prompt for *ckstr* is:

Enter an appropriate value [?,q]:

The default error message is dependent upon the type of validation involved. The user will be told either that the length or the pattern matching failed.

The default help message is also dependent upon the type of validation involved. If a regular expression has been defined, the message is:

Please enter a string which matches the following pattern: regexp

Other messages define the length requirement and the definition of a string.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valstr* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

cktime - display a prompt; verify and return a time of day

SYNOPSIS

cktime
[-Q] [-W *width*] [-f *format*] [-d *default*] [-h *help*]
[-e *error*] [-p *prompt*] [-k *pid*[-s *signal*]]

errtime
[-W] [-e *error*] [-f *format*]

helptime
[-W] [-h *help*] [-f *format*]

valtime
[-f *format*] *input*

DESCRIPTION

cktime prompts a user and validates the response. It defines, among other things, a prompt message whose response should be a time, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return). The user response must match the defined format for the time of day.

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *cktime* command. They are *errtime* (which formats and displays an error message), *helptime* (which formats and displays a help message), and *valtime* (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt. When *format* is defined in the *errtime* and *helptime* modules, the messages will describe the expected format.

OPTIONS

-Q
Specifies that *quit* will not be allowed as a valid response.

-W
Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-f
Specifies the format against which the input will be verified. Possible formats and their definitions

are:

%H	**=**	**hour (00 - 23)**
%I	=	hour (00 - 12)
%M	=	minute (00 - 59)
%p	=	ante meridian or post meridian
%r	=	time as %I:%M:%S %p
%R	=	time as %H:%M (the default format)
%S	=	seconds (00 - 59)
%T	=	time as %H:%M:%S

-d

Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h

Defines the help messages as *help*.

-e

Defines the error message as *error*.

-p

Defines the prompt message as *prompt*.

-k

Specifies that process ID *pid* is to be sent a signal if the user aborts with *quit*.

-s

Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input

Input to be verified against format criteria.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

4 = Garbled format argument

NOTES

The default prompt for *cktime* is:

Enter the time of day [?,q]:

The default error message is:

ERROR - Please enter the time of day, using the following format: <format>

The default help message is:

Please enter the time of day, using the following format: <format>

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valtime* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckuid - prompts for and validates a user ID

SYNOPSIS

ckuid

[-Q] [-W *width*] [-m] [-d *default*] [-h *help*] [-e *error*]□
[-p *prompt*] [-k *pid*[-s *signal*]]

erruid

[-W] [-e *error*]

helpuid

[-W] [-m] [-h *help*]

valuid

input

DESCRIPTION

ckuid prompts a user and validates the response. It defines, among other things, a prompt message whose response should be an existing user ID, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckuid* command. They are *erruid* (which formats and displays an error message), *helpuid* (which formats and displays a help message), and *valuid* (which validates a response). These modules should be used in conjunction with FML objects. In this instance, the FML object defines the prompt.

OPTIONS

-Q

Specifies that *quit* will not be allowed as a valid response.

-W

Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-m

Displays a list of all logins when help is requested or when the user makes an error.

-d

Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h

Defines the help messages as *help*.

-e

Defines the error message as *error*.

-p

Defines the prompt message as *prompt*.

-k

Specifies that process ID *pid* is to be sent a signal if the user abort with *quit*.

-s

Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input

Input to be verified against */etc/passwd*.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

NOTES

The default prompt for *ckuid* is:

Enter the login name of an existing user [?,q]:

The default error message is:

ERROR - Please enter the login name of an existing user.
Select the help option (?) for a list of valid login names.
(Last line appears only if the *-m* option of *ckuid* is used)

The default help message is:

Please enter the login name of an existing user.
(If the *-m* option of *ckuid* is used, a list of valid groups is also displayed.)

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valuid* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

ckyorn - prompts for and validates yes/no

SYNOPSIS

ckyorn
[-Q] [-W *width*] [-d *default*] [-h *help*] [-e *error*]
[-p *prompt*] [-k *pid*[-s *signal*]]

erryorn
[-W] [-e *error*]

helpyorn
[-W] [-h *help*]

valyorn
input

DESCRIPTION

ckyorn prompts a user and validates the response. It defines, among other things, a prompt message for a *yes* or *no* answer, text for help and error messages, and a default value (which will be returned if the user responds with a carriage return).

All messages are limited in length to 70 characters and are formatted automatically. Any white space used in the definition (including newline) is stripped. The *-W* option cancels the automatic formatting. When a tilde is placed at the beginning or end of a message definition, the default text will be inserted at that point, allowing both custom text and the default text to be displayed.

If the prompt, help or error message is not defined, the default message (as defined under NOTES) will be displayed.

Three visual tool modules are linked to the *ckyorn* command. They are *erryorn* (which formats and displays an error message), *helpyorn* (which formats and displays a help message), and *valyorn* (which validates a response). These modules should be used in conjunction with FACE objects. In this instance, the FACE object defines the prompt.

OPTIONS

-Q
Specifies that *quit* will not be allowed as a valid response.

-W
: Specifies that prompt, help and error messages will be formatted to a line length of *width*.

-d
: Defines the default value as *default*. The default is not validated and so does not have to meet any criteria.

-h
: Defines the help messages as *help*.

-e
: Defines the error message as *error*.

-p
: Defines the prompt message as *prompt*.

-k
: Specifies that process ID *pid* is to be sent a signal if the user aborts with *quit*.

-s
: Specifies that the process ID *pid* defined with the *-k* option is to be sent signal *signal* when *quit* is chosen. If no signal is specified, *SIGTERM* is used.

input
: Input to be verified as *y*, *yes*, *Y*, *Yes*, *YES* or *n*, *no*, *N*, *No*, *NO*.

EXIT CODES

0 = Successful execution

1 = EOF on input

2 = Usage error

3 = User termination (quit)

NOTES

The default prompt for *ckyorn* is:

Yes or No [y,n,?,q]:

The default error message is:

ERROR - Please enter yes or no.

The default help message is:

To respond in the affirmative, enter y, yes, Y, or YES.
To respond in the negative, enter n, no, N, or NO.

When the *quit* option is chosen (and allowed), *q* is returned along with the return code *3*. The *valyorn* module will not produce any output. It returns zero for success and non-zero for failure.

NAME

delsysadm - *sysadm* interface menu or task removal tool

SYNOPSIS

delsysadm
: task | [-r] *menu*

DESCRIPTION

The *delsysadm* command deletes a *task* or *menu* from the *sysadm* interface and modifies the interface directory structure on the target machine.

OPTIONS

task | *menu*
: The logical name and location of the menu or task within the interface menu hierarchy. Begin with the top menu *main* and proceed to where the menu or the task resides, separating each name with colons.

See EXAMPLES.

-r

If the *-r* option is used, this command will recursively remove all sub-menus and tasks for this menu. If the *-r* option is not used, the menu must be empty.

delsysadm should only be used to remove items added as "on-line" changes with the *edsysadm* command. Such an addition will have a package instance tag of ONLINE. If the task or menu (and its sub-menus and tasks) have any package instance tags other than ONLINE, you are asked whether to continue with the removal or to exit. Under these circumstances, you probably do not want to continue and you should rely on the package involved to take the necessary actions to delete this type of entry.

The command exits successfully or provides the error code within an error message.

EXIT CODES

0 = Successful execution

2 = Invalid syntax

3 = Menu or task does not exist

4 = Menu not empty

5 = Unable to update interface menu structure

EXAMPLES

To remove the *nformat* task, execute:

delsysadm main:applications:ndevices:nformat

SEE ALSO

edsysadm(1M), sysadm(1M)

NOTES

Any menu that was originally a placeholder menu (one that only appears if submenus exist under it) will be returned to placeholder status when a deletion leaves it empty.

When the *-r* option is used, *delsysadm* checks for dependencies before removing any subentries. (A dependency exists if the menu being removed contains an entry placed there by an application package). If a dependency is found, the user is shown a list of packages that depend on the menu being deleted and asked whether or not to continue. If the answer is yes, the menu and all of its menus and tasks are removed (even those shown to have dependencies). If the answer is no, the menu is not deleted.

delsysadm should only be used to remove menu or task entries that have been added to the interface with *edsysadm*.

NAME

edsysadm - *sysadm* interface editing tool

SYNOPSIS

edsysadm

DESCRIPTION

edsysadm is an interactive tool that adds or changes either menu and task definitions in the *sysadm* interface. It can be used to make changes directly on-line on a specific machine or to create changes that will become part of a software package. The command creates the administration files necessary to achieve the requested changes in the interface and either places them in the appropriate place for on-line changes or saves them to be included in a software package.

edsysadm presents several screens, first prompting for which type of menu item you want to change, *menu* or *task*, and then for what type of action to take, *add* or *change*. When you select *add*, a blank menu or task definition (as described below) is provided for you to fill in. When you select *change*, a series of screens is presented to help identify the definition you wish to change. The final screen presented is the menu or task

definition filled in with its current values, which you can then edit.

The menu definition prompts and their descriptions are:

Menu Name
: The name of the new menu (as it should appear in the lefthand column of the screen). This field has a maximum length of 16 alphanumeric characters.

Menu Description
: A description of the new menu (as it should appear in the righthand column of the screen). This field has a maximum length of 58 characters and can consist of any alphanumeric character except at sign (@), caret (^), tilde (~), back grave (`), grave ('), and double quotes (").

Menu Location
: The location of the menu in the menu hierarchy, expressed as a menu pathname. The pathname should begin with the main menu followed by all other menus that must be traversed (in the order they are traversed) to access this menu. Each menu name must be separated by colons. For example, the menu location for a menu entry being added to the Applications menu is *main:applications*. *Do not include the menu name in this location definition.* The complete pathname to this menu entry will be the menu location plus the menu name defined at the first prompt.

  This is a scrollable field, showing a maximum of 50 alphanumeric characters at a time.

Menu Help File Name
: Pathname to the item help file for this menu entry. If it resides in the directory from which you invoked *edsysadm*, you do not need to give a full pathname. If you name an item help file that does not exist, you are placed in an editor (as defined by $EDITOR) to create one. The new file is created in the current directory and named *Help*.

The task definition prompts and their descriptions are:

Task Name
: The name of the new task (as it should appear in the lefthand column of the screen). This field has a maximum length of 16 alphanumeric characters.

Task Description
: A description of the new task (as it should appear in the righthand column of the screen). This field has a maximum length of 58 characters and can consist of any alphanumeric character except at sign (@), caret (^), tilde (~), back grave (`), grave ('), and double quotes (").

Task Location
: The location of the task in the menu hierarchy, expressed as a pathname. The pathname should begin with the main menu followed by all other menus that must be traversed (in the order they are traversed) to access this task. Each menu name must be separated by colons. For example, the task location for a task entry being added to the applications menu is *main:applications*. *Do not include the task name in this location definition*. The complete pathname to this task entry will be the task location as well as the task name defined at the first prompt.

  This is a scrollable field, showing a maximum of 50 alphanumeric characters at a time.

Task Help File Name
: Pathname to the item help file for this task entry. If it resides in the directory from which you invoked *edsysadm*, you do not need to give a full pathname. If you name an item help file that does not exist, you are placed in an editor (as defined by $EDITOR) to create one. The new file is created in the current directory and named *Help*.

Task Action
: The FACE form name or executable that will be run when this task is selected. This is a scrollable field, showing a maximum of 58 alphanumeric characters at a time. This pathname can be relative to the current directory as well as absolute.

Task Files
: Any FACE objects or other executables that support the task action listed above and might be called from within that action. *Do not include the help file name or the task action in this list*. Pathnames can be

relative to the current directory as well as absolute. A dot (.) implies "all files in the current directory" and includes files in subdirectories.

This is a scrollable field, showing a maximum of 50 alphanumeric characters at a time.

Once the menu or task has been defined, screens for installing the menu or task or saving them for packaging are presented. The package creation or on-line installation is verified and you are informed upon completion.

SEE ALSO

delsysadm(1M), pkgmk(1M), prototype(4), sysadm(1M)

NOTES

For package creation or modification, this command automatically creates a menu information file and a *prototype* file in the current directory (the directory from which the command is executed). The menu information file is used during package installation to modify menus in the menu structure. A *prototype* file is an installation file which gives a listing of package contents.

The *prototype* file created by *edsysadm* lists the files defined under task action and gives them the special installation class of "admin". The contents of this *prototype* file must be incorporated in the package *prototype* file.

For on-line installation, *edsysadm* automatically creates a menu information file and adds or modifies the interface menu structure directly.

The item help file must follow the format shown in the appendix of [8].

NAME

idinstallf - add a driver component to the software installation database

SYNOPSIS

idinstallf

[-u|-f] [-c *class*] *pkginst module*

DESCRIPTION

idinstallf is called by driver software package (DSP) installation script. Components of a DSP are typically installed by the *idinstall* command. *idinstallf* examines the */etc/conf* directory for driver component files and adds these files to the installation database with type *f* and class *class* (default: *none*). Directories are stored asa implicit directories with the current attribute values of their pathnames.

OPTIONS

-f

Indicates that installation is complete. This option has the same effect as the *-f* option of the *installf* command.

-u

Displays a help text.

-c *class*

Class to which installed objects should be assigned. The default class is *none*.

pkginst

Name of the package instance with which the driver component pathnames should be associated. *pkginst* is added to the list of packages in */etc/conf/cf.d/mdevice*.

module

Name used as a directory name under */etc/conf/pack.d* and as a file name under the corresponding */etc/conf* directories.

idinstallf has the same effect as the *installf* applied to each of the available driver components.

There is no *idremovef* command corresponding to *removef*.

SEE ALSO

idinstall(1M), installf(1M), removef(1M), mdevice(4)

NAME

installf - add a file to the software installation database

SYNOPSIS

installf

[-c *class*] *pkginst pathname*[*ftype*[[*major minor*][*mode owner group*]]

installf

[-c *class*] *pkginst* -

installf

-f[-c *class*] *pkginst*

DESCRIPTION

installf informs the system that a pathname not listed in the *pkgmap* file is being created or modified. *installf* should be invoked before any file modifications have occurred.

When the second synopsis is used, the pathname descriptions will be read from standard input. These descriptions are the same as would be given in the first synopsis but the information is given in the form of a list. The descriptions should be in the form:

pathname [ftype [[major minor] [mode owner group]]

After all files have been appropriately created and/or modified, *installf* should be invoked with the *-f* synopsis to indicate that installation is final. Links will be created at this time and, if attribute information for a pathname was not specified during the original invocation of *installf* or was not already stored on the system, the current attribute values for the pathname will be stored. Otherwise, *installf* verifies that attribute values match those given on the command line, making corrections as necessary. In all cases, the current content information is calculated and stored appropriately in the software installation database.

OPTIONS

-c *class*

Class to which installed objects should be associated. Default class is *none*.

pkginst

Name of package instance with which the pathname should be associated.

pathname

Pathname that is being created or modified.

ftype

A one-character field that indicates the file type. Possible file types include:

f - a standard executable or data file

e - a file to be edited upon installation or removal

v - volatile file (one whose contents are expected to change)

d - directory

x - an exclusive directory

l - linked file

p - named pipe

c - character special device

b - block special device

s - symbolic link

major

The major device number. The field is only specified for block or character special devices.

minor

The minor device number. The field is only specified for block or character special devices.

mode

The octal mode of the file (for example, 0664). A question mark (?) indicates that the mode will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked or symbolically linked files.

owner

The owner of the file (for example, *bin* or *root*). The field is limited to 14 characters in length. A question mark (?) indicates that the owner will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked or symbolically linked files.

group

The group to which the file belongs (for example, *bin* or *sys*). The field is limited to 14 characters in length. A question mark (?) indicates that the group will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked or symbolically linked files.

-f

Indicates that installation is complete. This option is used with the final invocation of *installf* (for all files of a given class).

EXAMPLE

The following example shows the use of *installf* invoked from an optional *preinstall* or *postinstall* script:

```
#□create□/dev/xt□directory
#□(needs□to□be□done□before□drvinstall)
installf□$PKGINST□/dev/xt□d□755□root□sys□||
□□□□□□exit□2
majno=`/usr/sbin/drvinstall□-m□/etc/master.d/xt
□□□□□-d□$BASEDIR/data/xt.o□-v1.0`□||
□□□□□□exit□2
i=00
while□[□$i□-lt□$limit□]
do
□□□□□□□□for□j□in□0□1□2□3□4□5□6□7
□□□□□□□□do
□□□□□□□□□□□□□□□□□□echo□/dev/xt$i$j□c□$majno□`expr□$i□?□8□+□$j`
□□□□□□□□□□□□□□□□□□□□□□□□□□□□644□root□sys□|
□□□□□□□□□□□□□□□□□□echo□/dev/xt$i$j=/dev/xt/$i$j
□□□□□□□□done
□□□□□□□□i=`expr□$i□+□1`
□□□□□□□□[□$i□-le□9□]□&&□i="0$i"□#□add□leading□zero
done□|□installf□$PKGINST□-□||□exit□2
#□finalized□installation,□create□links
installf□-f□$PKGINST□||□exit□2
```

SEE ALSO

pkgadd(1M), pkgask(1M), pkgchk(1M), pkginfo(1M), pkgmk(1M), pkgparam(1M), pkgproto(1M), pkgtrans(1M), pkgrm(1M), removef(1M)

NOTES

When *ftype* is specified, all applicable fields, as shown below, must be defined:

```
ftype□□□□□□□□□□□□□□□Required□fields
p□x□d□f□v□or□e□□□□mode□owner□group
c□or□b□□□□□□□□□□□□□□mode□□owner□□group
```

The *installf* command will create directories, named pipes and special devices on the original invocation. Links are created when *installf* is invoked with the *-f* option to indicate installation is complete.

Links should be specified as *path1=path2*. *path1* indicates the destination and *path2* indicates the source file.

Files installed with *installf* will be placed in the class *none*, unless a class is defined with the command. Subsequently, they will be removed when the associated package is deleted. If this file should not be deleted at the same time as the package, be certain to assign it to a class which is ignored at removal time. If special action is required for the file before removal, a class must be defined with the command and an appropriate class action script delivered with the package.

When classes are used, *installf* must be used as follows:

installf -c class1 ...
installf -f -c class1 ...
installf -c class2 ...
installf -f -c class2 ...

NAME

pkgadd - install software packages

SYNOPSIS

pkgadd
[-d *device*] [-r *response*] [-n] [-a *admin*]□
[*pkginst1*[*pkginst2*[...]]]

pkgadd
[-s *spool*[-d *device*] [*pkginst1*[*pkginst2*[...]]]

DESCRIPTION

pkgadd installs the contents of a software package from a data medium or directory. Used without the *-d* option, *pkgadd* looks for the package in the default spool directory (*/var/spool/pkg*). Used with the *-s* option, it transfers the package to the default spool directory instead of installing it.

OPTIONS

-d
Installs or copies a package from *device*. *device* can be a full path name to a directory or the identifiers for tape, floppy disk or removable disk (for example, */var/tmp*, */dev/dsk/f0*, or */dev/dsk/f1*). It can also be the device alias.

-r
Identifies a file or directory, *response*, which contains output from a previous *pkgask* session. This file supplies the interaction responses that would be requested by the package in interactive mode. *response* must be a full pathname.

-n
Installation occurs in non-interactive mode. The default mode is interactive.

-a
Defines an installation administration file, *admin*, to be used in place of the default administration file. The token *none* overrides the use of any *admin* file, and thus forces interaction with the user. Unless a full path name is given, *pkgadd* looks in the */var/sadm/install/admin* directory for the file.

pkginst
Specifies the package or list of packages to be installed. The token *all* may be used to refer to all packages available on the data medium or directory.

-s
Transfers the package into the default spool directory instead of installing it.

-l
Log error messages in /var/sadm/install/logs.

-q
No messages (quiet mode).

-O

The selection menu for the package instance is only displayed once.

When executed without options, *pkgadd* uses */var/spool/pkg* (the default spool directory).

DIAGNOSTICS

There are the following return codes:

Code	Meaning
0	Execution successful
1	Aborted (usually incorrect call)
2	Warning (e.g. installed only partially)
3	Interrupt by user
4	Aborted due to admin entry
5	Aborted: interaction required
+10	Reboot necessary after installation of all packages
+20	Immediate reboot required
NOSET(77)	No package selected (with Set Installation Packages only)
>=79	Aborted by pkg-internal error:
	Device errors:
79	Device not found
80	Device not found in device table
	Datastream errors:
81	No archive in datastream
82	Initialisation of datastream device not possible
83	Error in ds_next()
84	Datastream expected
	Package errors:
85	Empty package list
86	pkgmap or pkginfo missing
87	Response file cannot be created/deleted
88	Error when opening file□ pkgmap/pkginfo/depend/admin
89	Error when opening response file
90	Package parts <1
91	Incorrect processing of pkgmap
92	Damaged file in package
93	Damaged directory structure
94	EXISTING_VERSION of the package not installed
95	EXISTING_LOAD of the package not installed
96	Inconsistent pkgmap entries and contents entries

97	Undefined CLASSES parameter
	System errors:
99	No more memory
100	ulimit error
101	Unidentifiable working directory
102	Pipe error
103	System run state cannot be determined
104	pkginfo error
105	uncompress error
106	Lock error
107	umount not possible
108	fseek() error
	Error in contents file:
110	Error when opening contents file
111	Incorrect entry in contents file
112	Incorrect entry status in contents file
113	Write error in contents file
	Directory error:
116	Directory cannot be created
117	Changing to directory not possible
118	Copying to directory not possible
119	Deleting directory not possible
120	Temporary file cannot be created
	Pyramid function errors:
121	Error in _pyr_save_cp
122	Error in _pyr_copy_files
123	Error in _pyr_arm

NOTES

When transferring a package to a spool directory, the *-r*, *-n*, and *-a* options cannot be used.

The *-r* option can be used to indicate a directory name as well as a filename. The directory can contain numerous *response* files, each sharing the name of the package with which it should be associated. This would be used, for example, when adding multiple interactive packages with one invocation of *pkgadd*. Each package would need a *response* file. If you create response files with the same name as the package (i.e. *package1* and *package2*), then name the directory in which these files reside after the *-r*.

The *-n* option will cause the installation to halt if any interaction is needed during installation.

NAME

pkgask - stores answers to a request script

SYNOPSIS

/usr/sbin/pkgask

[-d *device*] -r *response pkginst*[*pkginst*[...]]

DESCRIPTION

pkgask allows the administrator to store answers to an interactive package (one with a request script). Invoking this command generates a *response* file that can then be used as input at installation time. The use of this *response* file prevents any interaction from occurring during installation since the file already contains all of the information the package needs.

OPTIONS

-d

Runs the request script for a package on *device*. *device* can be a directory pathname or the identifiers for tape, floppy disk or removable disk (for example, */var/tmp*, */dev/dsk/0s2*, and */dev/dsk/f0t*). The default device is the installation spool directory.

-r

Identifies a file or directory, which should be created to contain the responses to interaction with the package. The name must be a full pathname. The file, or directory of files, can later be used as input to the *pkgadd* command.

pkginst

Specifies the package or list of packages for which interactive responses will be created. The token *all* may be used to refer to all packages available on the data medium or directory.

SEE ALSO

installf(1M), pkgadd(1M), pkgchk(1M), pkgmk(1M), pkginfo(1M), pkgparam(1M), pkgproto(1M), pkgtrans(1M), pkgrm(1M), removef(1M)

NOTES

The *-r* option can be used to indicate a directory name as well as a filename. If several packages are named, a directory name must be used. In this directory *pkgask* sets up one file for each package, giving it the same name as the package. This file contains the interactive responses for the corresponding package.

NAME

pkgchk - check accuracy of installation

SYNOPSIS

/usr/sbin/pkgchk

[-l|-acfqv] [-nx] [-p *path1*[,*path2* ...] [-i *file*]□
[*pkginst* ...]

/usr/sbin/pkgchk

-d *device*[-l|v] [-p *path1*[,*path2* ...] [-i file]□
[*pkginst* ...]

/usr/sbin/pkgchk

[-m *pkgmap*[-e *envfile*] [-l|-acfqv] [-nx] [-i *file*]□
[-p *path1*[,*path2* ...]]

DESCRIPTION

pkgchk checks the accuracy of installed files or, by use of the *-l* option, displays information about package files. The command checks the integrity of directory structures and the files. Discrepancies are reported on *stderr* along with a detailed explanation of the problem.

The first synopsis is used to list or check the contents and/or attributes of objects that are currently installed on the system. Package names may be listed on the command line, or by default the entire contents of a machine will be checked.

The second synopsis is used to list or check the contents of a package which has been spooled on the specified device, but not installed. Note that attributes cannot be chekked for spooled packages.

The third synopsis is used to list or check the contents and/or attributes of objects which are described in the indicated *pkgmap*.

OPTIONS

-l
: Lists information on the selected files that make up a package. It is not compatible with the *a*, *c*, *f*, *g*, and *v* options.

-a
: Audits the file attributes only, does not check file contents. Default is to check both.

-c
: Audits the file contents only, does not check file attributes. Default is to check both.

-f
: Corrects file attributes if possible. When *pkgchk* is invoked with this option it creates directories, named pipes, links and special devices if they do not already exist. If used with the *-x* option, it removes all files which are in an exclusive directory but do not appear in the software installation database.

-q
: Quiet mode. Does not give messages about missing files.

-v
: Verbose mode. Files are listed as processed.

-n
: Does not check volatile or editable files. This should be used for most post-installation checking.

-N
: Always checks volatile or editable files.

-x
: Searches exclusive directories, looking for files which exist that are not in the installation software database or the indicated *pkgmap* file.

-p
: Only checks the accuracy of the pathname or pathnames listed. *pathname* can be one or more pathnames separated by commas (or by white space, if the list is quoted).

-i
: Reads a list of pathnames from *file* and compares this list against the installation software database or the indicated *pkgmap* file. Pathnames which are not contained in *inputfile* are not checked.

-d
: Specifies the device on which a spooled package resides. *device* can be a directory pathname or the identifier for a data medium (for example, */var/tmp* or */dev/diskette*).

-m
: Checks the package against the software installation database (or *pkgmap* file) *pkgmap*.

-e
: Indicates that the package parameters are to be taken from the *pkginfo* file *envfile be used to resolve parameters noted in the specified pkgmap file*.

pkginst
: Specifies the package instance or instances to be checked. The format *pkginst.** can be used to check all instances of a package. The default is to display or check all information about all installed packages.

SEE ALSO

pkgadd(1M), pkgask(1M), pkginfo(1M), pkgrm(1M), pkgtrans(1M)

NAME

pkginfo - display software package information

SYNOPSIS

pkginfo
[-q|x|l][-p|i] [-a *arch*] [-v *version*] [-c *category1*,□
[*category2*[, ...]]][, *pkginst*[, *pkginst*[. ...]]]

pkginfo
[-d *device* [-q|x|l][-a *arch*][-v *version*][-c *category1*,□
[*category2*[, ...]]][, *pkinst*[, *pkinst*[. ...]]]]

DESCRIPTION

pkginfo displays information about software packages which are installed on the system (with the first synopsis) or which reside on a particular device or directory (with the second synopsis).

OPTIONS

-q
Does not list any information, but is simply used to check whether or not a package has been installed.

-x
Designates an extracted listing of package information. It contains the packagae abbreviation, package name, package architecture (if available) and package version (if available).

-l
Designates long format, which includes all available information about the designated package(s).

-p
Designates that information should presented only for partially installed packages.

-i
Designates that information should presented only for fully installed packages.

-a
Designates that information should presented only for packages with an architecture of *arch*.

-v
Specifies the version of the package as *version*. "All compatible versions" can be requested by preceding the version name with a tilde (~). Multiple white space is replaced with a single space during version comparison.

-c
Selects packages to display based on the category *category*. (Categories are defined in the category field of the *pkginfo* file.) If more than one category is supplied, the package must only match one of the list of categories. The match is not case-specific.

pkginst
Designates a package by its instance. An instance can be the package abbreviation or a specific instance (for example, *inst.1* or *inst.beta*). All instances of a packagae can be requested by *inst.**.

-d
Defines a device, *device*, on which the software resides. *device* can be a direcetory pathname or the identifier for a data medium. The special keyword "spool" may be used to indicate the default installation spool directory.

-L
pkginst name only.

-N
No pre-SVR4 packages.

SEE ALSO

pkgadd(1M), pkgask(1M), pkgchk(1M), pkgrm(1M), pkgtrans(1M).

NOTES

Without options, *pkginfo* lists the primary category, package instance, and name of all completely installed and

partially installed packages. One line is produced per package selected.

The *-p* and *-i* options are meaningless if used in conjunction with the *-d* option.

The *-q*, *-x* and *-l* are mutually exclusive.

NAME

pkgmk - produce an installable package

SYNOPSIS

pkgmk

[-o] [-d *device*] [-r *rootpath*] [-b *basdir*] [-l *limit*]□
[-a *arch*] [-v *version*] [-p *pstamp*] [-f *prototype*] □
[*variable=value ...*] [*pkginst*]

DESCRIPTION

pkgmk produces an installable package. The package is stored in file system format (see *pkgtrans(1M)).*

The command uses the package *prototype* file as input and creates a *pkgmap* file. The contents for each entry in the *prototype* file is copied to the appropriate output location. Information concerning the contents (checksum, file size, modification date) is computed and stored in the *pkgmap* file, along with attribute information specified in the *prototype* file.

Pathnames in a call to *pkgmk* must be in absolute form.

OPTIONS

-o

Overwrite mode. Package instance will be overwritten if it already exists.

-c

The package is compressed with the compress(1) command. Only information files are not compressed.

-O

Optimize compression with -c: The files are compressed when being copied to the output medium. Otherwise compression is a separate process. -O cannot be used for divided packages.

-d

Creates the package on *device*. *device* can be a directory pathname or the identifiers for a data medium. The default device is the installation spool directory.

-r

Prepends the indicated pathname *rootpath* to objects with absolute pathnames to locate them on the source machine.

-b

Prepends the indicated *basedir* to locate relocatable objects on the source machine.

-l

Specifies the maximum size in 512 byte blocks of the output device as *limit*. If the output file is a directory or a mountable device, *pkgmk* will automatically calculate the amount of available space on the output device. This options is useful if the *pkgtrans* command is to be used later to distribute the package over multiple (smaller) volumes.

-a

Overrides the value of ARCH* in the *pkginfo* file with *arch*.

-v

Overrides the value of VERSION* in the *pkginfo* file with *version*.

-p

Overrides the value of PSTAMP* in the *pkginfo* file with *pstamp*.

-f

Uses the file *prototype* as input to the command. The default *prototype* filename is *[Pp]rototype*.

variable=value

The variable *variable* is given the value *value* for the duration of package generation. This assignment is not used within the completed package.

pkginst

Specifies the package by its instance. An instance can be the package abbreviation or a specific instance (for example, *inst.1*).

SEE ALSO

pkgparam(1M), pkgproto(1M), pkgtrans(1M).

NOTES

Architecture information is provided on the command line with the *-a* option or in the *prototype* file. If no architecture information is supplied at all, the output of *uname -m* will be used.

Version information is provided on the command line with the *-v* option or in the *prototype* file. If no version information is supplied, a default based on the current date will be provided.

Command line definitions for both architecture and version override the *prototype* definitions.

NAME

pkgparam - displays package parameter values

SYNOPSIS

pkgparam

[-v] [-d *device*] *pkginst* [*param*[...]-]

pkgparam

-f *file* [-v] [*param*[...]-]

DESCRIPTION

pkgparam displays the value of the software package parameter or parameters specified on the command line. The values are taken from the package's *pkginfo* file (with the first synopsis) or from the specific file named with the *-f* option (with the second synopsis).

One parameter value is shown per line. Only the value of a parameter is given unless the *-v* option is used. With the *-v* option, the output of the command is in this format:

parameter1='value1'
parameter2='value2'
parameter3='value3'

If no parameters are specified on the command line, values for all parameters associated with the package are shown.

OPTIONS

-v

Changes the output format. Displays name of parameter and its value.

-d

Specifies the *device* on which a *pkginst* is stored. It can be a directory pathname or the identifier for a data medium (for example, */var/tmp*, */dev/dsk/f0t*). If neither the *-d* option nor the *-f* option is specified, the installed package *pkginst* is read.

-f

Requests that the command read the file *file* for parameter values.

pkginst

Defines a specific package (or package instance) for which parameter values should be displayed.

param

Defines a specific parameter whose value should be displayed.

SEE ALSO

pkgmk(1M), pkgparam(3x), pkgproto(1M), pgktrans(1M)

ERRORS

If parameter information is not available for the indicated package, the command exits with a non-zero status.

NOTES

The *-f* synopsis allows you to specify the file from which parameter values should be extracted. This file should be in the same format as a *pkginfo* file. As an example, such a file might be created during package development and used while testing software during this stage.

NAME

pkgproto - generate a prototype file

SYNOPSIS

pkgproto
[-i] [-c *class*] [*path1*[=*path2*] ...]

DESCRIPTION

pkgproto scans the indicated paths and generates a prototype file that may be used as input to the *pkgmk* command.

Pathnames in a call to *pkgmk* must be in absolute form.

OPTIONS

-i
With symbolic links, the target file is used instead of the link.

-c
Maps the class of all paths to *class*.

-n
When called with *path1=path2* only *path2* is entered in prototype (except links).

-x
Size, check sum and time of modification are appended to each entry, i.e. the output is the similar to pkgmap.

path1
Pathname where objects are located.

path2
Pathname which should be substituted on output for *path1*.

If no paths are specified on the command line, standard input is assumed to be a list of paths. If the pathname listed on the command line is a directory, the contents of the directory is searched. However, if input is read from *stdin*, a directory specified as a pathname will not be searched.

EXAMPLES

The following two examples show uses of pkgproto and a parial listing of the output produced.

Example 1, pkgproto(1M):

```
$ pkgproto /usr/bin=bin /usr/usr/bin=usrbin /etc=etc
f none bin/sed=/bin/sed 0775 bin bin
f none bin/sh=/bin/sh 0755 bin daemon
f none bin/sort=/bin/sort 0755 bin bin
f none usrbin/sdb=/usr/bin/sdb 0775 bin bin
f none usrbin/shl=/usr/bin/shl 4755 bin bin
d none etc/master.d 0755 root daemon
f none etc/master.d/kernel=/etc/master.d/kernel 0644 root daemon
f none etc/rc=/etc/rc 0744 root daemon
```

Example 2, pkgproto(1M):

```
$ find / -type d -print | pkgproto
d none / 755 root root
d none /usr/bin 755 bin bin
d none /usr 755 root root
d none /usr/bin 775 bin bin
d none /etc 755 root root
d none /tmp 777 root root
```

SEE ALSO

pkgmk(1M), pkgparam(1M), pkgtrans(1M)

NOTES

By default, *pkgproto* creates symbolic link entries for any symbolic link encountered (*ftype=s*). When you use the *-i* option, *pkgproto* creates a file entry for symbolic links (*ftype=f*).

The *prototype* file would have to be edited to assign such file types as "v" (volatile), "e" (editable), or "x" (exclusive directory). *pkgproto* detects linked files. If multiple files are linked together, the first path encountered is considered the source of the link.

NAME

pkgtrans - translate package format

SYNOPSIS

pkgtrans
[-ions] *device1 device2*[*pkginst1*[*pkginst2*[...]]]

DESCRIPTION

pkgtrans translates an installable package from one format to another. It translates:

- a file system format to a datastream
- a datastream to a file system format
- a file system format to another file system format

OPTIONS

-i
Copies only the *pkginfo* and *pkgmap* files.

-o
Overwrites an existing package on the destination device.

-n
Creates a new instance if any instance of this package already exists.

-s
Indicates that the package should be written to *device2* as a datastream rather than as a file system. The default behavior is to write a file system format on devices that support both formats.

device1
Indicates the source device. The package or packages on this device will be translated and placed on *device2*.

device2
Indicates the destination device. Translated packages will be placed on this device.

pkginst
Specifies which package instance or instances on *device1* should be translated. The token *all* may be used to indicate all packages. *pkginst.** can be used to indicate all instances of a package. If no packages are defined, *pkgtrans* shows all packages on the device *device1* and asks which to translate.

EXAMPLES

The following example translates all packages on the floppy drive */dev/diskette* and places the translations on */tmp*.

pkgtrans /dev/diskette /tmp all

The next example translates packages *pkg1* and *pkg2* on */tmp* and places their translations (i.e., a datastream) on the *9track1* output device.

pkgtrans /tmp 9track1 pkg1 pkg2

The next example translates *pkg1* and *pkg2* on *tmp* and places them on the diskette in a datastream format.

pkgtrans -s /tmp /dev/diskette pkg1 pkg2

SEE ALSO

installf(1M), pkgadd(1M), pkgask(1M), pkginfo(1M), pkgmk(1M), pkgparam(1M), pkgproto(1M), pkgrm(1M), removef(1M).

NOTES

Device specifications can be either the special node name (*/dev/diskette1*) or the device alias (*diskette1*). The device *spool* indicates the default spool directory. Source and destination devices may not be the same.

By default, *pkgtrans* will not transfer any instance of a package if any instance of that package already exists on the destination device. Use of the *-n* option will create a new instance if an instance of this package already exists. Use of the *-o* option will overwrite the same instance if it already exists. Neither of these options are useful if the destination device is a datastream.

NAME

pkgrm - removes a package from the system

SYNOPSIS

pkgrm
[-n] [-a *admin*] [*pkginst1*[*pkginst2*[...]]]

pkgrm
-s *spool*[*pkginst*]

DESCRIPTION

pkgrm will remove a previously installed or partially installed package from the system. A check is made to determine if any other packages depend on the one being removed. The action taken if a dependency exists is defined in the installation administration file.

The default mode for the command is interactive mode, meaning that prompt messages are given during processing to allow the administrator to confirm the actions being taken. Non-interactive mode can be requested with the *-n* option.

The *-s* option can be used to specify the directory from which spooled packages should be removed.

OPTIONS

-n
Non-interactive mode.
If there is a need for interaction, the command will exit. Use of this option requires that at least one package instance be named upon invocation of the command.

-a
Defines an installation administration file, *admin*, to be used in place of the default installation administration file.

-s
Removes the specified package(s) from the directory *Spool*.

pkginst
Specifies the package to be removed. The format *pkg_abbrev.** can be used to remove all instances of

a package.

SEE ALSO

installf(1M), pkgadd(1M), pkgask(1M), pkgchk(1M), pkginfo(1M), pkgmk(1M), pkgparam(1M), pkgproto(1M), pkgtrans(1M), removef(1M)

NAME

removef – remove a file from software database

SYNOPSIS

removef
pkginst path1[*path2* ...]

removef
–f *pkginst*

DESCRIPTION

removef informs the system that the user, or software, intends to remove a pathname. Output from *removef* is the list of input pathnames that may be removed without collisions or errors (no other packages have a dependency on them).

After all files have been processed, *removef* should be invoked with the *–f* option to indicate that the removal phase is complete.

OPTIONS

-f
Indicates that the removal phase is complete.

pkginst
Defines the package instance with which the pathname(s) is (are) to be associated.

path
The pathname(s) to be removed.

EXAMPLE

The following shows the use of *removef* in an optional *preinstall* script:

```
echo "Folgende Dateien sind nicht laenger Teil dieses Pakets
□□□und werden daher geloescht."
removef $PKGINST /dev/xt[0-9][0-9][0-9] |
while read pathname
do
□□□echo "$pathname"
□□□rm -f $pathname
done
removef -f $PKGINST || exit 2
```

SEE ALSO

installf(1M), pkgadd(1M), pkgask(1M), pkgchk(1M), pkginfo(1M), pkgmk(1M), pkgproto(1M), pkgtrans(1M), pkgparam(3X)

NAME

admin - installation defaults file

DESCRIPTION

admin is a generic name for an ASCII file that defines default installation actions by assigning values to installation parameters. For example, it allows administrators to define how to proceed when the package being installed already exits on the system.

/var/sadm/install/admin/default is the default *admin* file delivered with. The default file is not writable, so to assign

values different from this file, create a new *admin* file. There are no naming restrictions for admin files. Name the file when installing a package with the *-a* option of *pkgadd*. If the *-a* option is not used, the default *admin* file is used.

Each entry in the *admin* file is a line that establishes the value of a parameter in the following form:

param=value

12 parameters can be defined in an *admin* file. A file is not required to assign values to all 12 parameters. If a value is not assigned, pkgadd asks the installer how to proceed.

The 12 parameters and their possible values are shown below except as noted. They may be specified in any order. Any of these parameters can be assigned the value *ask*, which means that if the situation occurs the installer is notified and asked to supply instructions at that time.

basedir
: Defines the base directory in which collectively relocatable objects should be installed. The following specifications for value are supported:

  dir
  : The base directory is directly specified. The value can contain the parameter *$PKGINST* to specify a base directory which is independent of the program package concerned. *dir* must be an absolute pathname, i.e. it must begin with "/".

  default
  : The base directory for collectively relocatable objects is retrieved (if present) from the *pkginfo(4)* file of the package to be installed.

  ask (or empty)
  : The base directory for collectively relocatable objects is requested interactively (if interaction is permitted).

  Prerequisites for *basedir* becoming effective are generally

  - the existence of freely relocatable objects
  - the presetting of *$BASEDIR* variables in the *pkginfo(4)* file with a non-zero standard value.

mail
: Defines a list of users to whom mail should be sent following installation of a package. If the list is empty, no mail is sent. If the parameter is not present in the admin file, the default value of root is used. The *ask* value cannot be used with this parameter.

runlevel
: Indicates resolution if the run level is not correct for the installation or removal of a package. The following specifications for value are supported:

  nocheck
  : Do not check for run level.

  quit
  : Abort installation if run level is not met.

conflict
: Specifies what to do if an installation expects to overwrite a previously installed file, thus creating a conflict between packages. The following specifications for value are supported:

  nocheck
  : Do not check for conflict; files in conflict will be overwritten.

  quit
  : Abort installation if conflict is detected.

  nochange
  : Override installation of conflicting files; they will not be installed.

setuid
: Checks for executables which will have setuid or setgid bits enabled after installation. The following

specifications for value are supported:

nocheck
: Do not check for setuid executables.

quit
: Abort installation if setuid processes are detected.

nochange
: Override installation of setuid processes; processes will be installed without setuid bits enabled.

action
: Determines if action scripts provided by package developers contain possible security impact. The following specifications for value are supported:

nocheck
: Ignore security impact of action scripts.

quit
: Abort installation if action scripts may have a negative security impact.

partial
: Checks to see if a version of the package is already partially installed on the system. The following specifications for value are supported:

nocheck
: Do not check for a partially installed package.

quit
: Abort installation if a partially installed package exists.

instance
: Determines how to handle installation if a previous version of the package (including a partially installed instance) already exists. The following specifications for value are supported:

quit
: Exit without installing if an instance of the package already exists (does not overwrite existing packages).

overwrite
: Overwrite an existing package if only one instance exists. If there is more than one instance, but only one has the same architecture, it overwrites that instance. Otherwise, the installer is prompted with existing instances and asked which to overwrite.

unique
: Do not overwrite an existing instance of a package. Instead, a new instance of the package is created. The new instance will be assigned the next available instance identifier.

idepend
: Controls resolution if other packages depend on the one to be installed. The following specifications for value are supported:

nocheck
: Do not check package dependencies.

quit
: Abort installation if package dependencies are not met.

rdepend
: Controls resolution if other packages depend on the one to be removed. The following specifications for value are supported:

nocheck
: Do not check package dependencies.

quit

Abort removal if package dependencies are not met.

space
: Controls resolution if disk space requirements for package are not met. The following specifications for *value* are supported:

  nocheck
  : Do not check space requirements (installation fails if it runs out of space).

  quit
  : Abort installation if space requirements are not met.

list_files
: nocheck - Processed files are not listed. Otherwise yes.

NOTES

The value *ask* should not be defined in an *admin* file that will be used for non-interactive installation (since by definition, there is no installer interaction). Doing so causes installation to fail when input is needed.

EXAMPLE

basedir=default
runlevel=quit
conflict=quit
setuid=quit
action=quit
partial=quit
instance=unique
idepend=quit
rdepend=quit
space=quit
list_files=nocheck

SEE ALSO

pkginfo(4).

NAME

compver - compatible versions file

DESCRIPTION

compver is an ASCII file used to specify previous versions of the associated package which are upward compatible. It is created by a package developer.

Each line of the file specifies a previous version of the associated package with which the current version is backward compatible.

Since some packages may require installation of a specific version of another software package, compatibility information is extremely crucial. Consider, for example, a package called "A" which requires version "1.0" of application "B" as a prerequisite for installation. If the customer installing "A" has a newer version of "B" (version 1.3), the *compver* file for "B" must indicate that "1.3" is compatible with version "1.0" in order for the customer to install package "A".

EXAMPLE

A sample *compver* file is shown below.

Version 1.3
Version 1.0

NOTES

The comparison of the version string disregards white space and tabs. It is performed on a word-by-word basis. Thus

"Version□□□□□□□1.3"□and□"Version□1.3"

would be considered the same.

NAME

copyright – copyright information file

DESCRIPTION

The *copyright* file is an *ASCII* file used to provide a copyright notice for a package. The text may be in any format. The full file contents (including comment lines) is displayed on the terminal at the time of package installation.

If a there is a copyright file for a package, it must identified as an information file in the package's *prototype* file.

SEE ALSO

prototype(4)

NAME

depend - software package dependencies file

DESCRIPTION

depend is an ASCII file used to specify information concerning dependencies a particular software package and other software packages. The file is created by the package developer.

Each entry in the *depend* file describes a dependency with one software package. The format of an entry is:

type pkg name
□□□□□□□[(arch)]version
□□□□□□□[(arch)]version
□□□□□□□...

The fields are:

type
Defines the dependency type. Must be one of the following characters:

P -
Indicates a prerequisite for installation, for example, the referenced package or versions must be installed.

I -
Implies that the existence of the indicated package or version is incompatible.

R -
Indicates a reverse dependency. Instead of defining the package's own dependencies, this designates that another package depends on this one. This type should be used only when an old package does not have a *depend* file but it relies on the newer package nonetheless. Therefore, the present package should not be removed if the designated old package is still on the system since, if it is removed, the old package will no longer work.

pkg
Indicates the package abbreviation.

name
Specifies the full package name.

[(*arch*)] *version* or [(*arch*)] *~version*
Specifies a particular package instance to which the dependency applies. The instance is identified by the package version and where necessary by the package architecture (in parentheses). Each instance specification must begin on a new line that begins with a blank or tab. A null version set equates to any version of the indicated package.

If the version specification in the *depend* file of a dependent package is preceded by a tilde (~), the

compver file is examined as well. That means that the *compver* ffile must be maintained in the package whose version is changing, and the new version must be compatible with the earlier version (see also Section "The depend file").

EXAMPLE

Here is an example of the relationship between the *depend* file and the *compver* file:

Package A *compver*file:

version 1.0
version 2.0
version 3.0

This means that VERSION in *pkginfo* may be followed by 1.0 - 3.0. These versions are compatible.

Package B *depend* file:

P□<package_a_abbreviation>□<full_package_name>
□□□□~□version□1.0

In this case package A's *compver* file will be examined when package B's dependencies are evaluated.

NAME

pkginfo -□□package characteristics file

DESCRIPTION

pkginfo is an *ASCII* file that describes the characteristics of the package along with information that helps control the flow of installation. It is created by the software package developer.

Each entry in the *pkginfo* file is a line that establishes the value of a parameter in the following form:

PARAM="value"

There is no required order in which the parameters must be specified within the file. Each parameter is described below. Only fields marked with an asterisk are mandatory.

*PKG**

Abbreviation for the package being installed, generally three characters in length (for example, *dir* or *pkg*). All characters in the abbreviation must be alphanumeric and the first may not be numeric. The abbreviation is limited to a maximum length of nine characters. *install*, *new*, and *all* are reserved abbreviations. *PKG* is a mandatory parameter.

*NAME**

Text that specifies the package name (maximum length of 256 ASCII characters). *NAME* is a mandatory parameter.

*ARCH**

A comma-separated list of alphanumeric tokens that indicate the architecture (for example, *i386*) associated with the package. The maximum length of a token is 16 characters and it cannot include a comma. This value can be specified with the *-a* option of the *pkgmk* command when the package is being built. *ARCH* is a mandatory parameter.

*VERSION**

Text that specifies the current version associated with the software package. The maximum length is 256 ASCII characters and the first character cannot be a left parenthesis. This value can be specified with the *-v* option of the *pkgmk* command when the package is being built. *VERSION* is a mandatory parameter.

*CATEGORY**

A comma-separated list of categories under which a package may be displayed. A package must at least belong to the system or application category. Categories are case-insensitive and may contain only alphanumerics. Each category is limited in length to 16 characters. *CATEGORY* is a mandatory parameter.

DESC

Text that describes the package (maximum length of 256 ASCII characters).

VENDOR
Used to identify the vendor that holds the software copyright (maximum length of 256 ASCII characters).

HOTLINE
Phone number and/or mailing address where further information may be received or bugs may be reported (maximum length of 256 ASCII characters).

EMAIL
An electronic address where further information is available or bugs may be reported (maximum length of 256 ASCII characters).

VSTOCK
The vendor stock number, if any, that identifies this product (maximum length of 256 ASCII characters).

CLASSES
A space-separated list of classes defined for a package. The order of the list determines the order in which the classes are installed. Classes listed first will be installed first (on a media by media basis). This parameter may be modified by the request script.

ISTATES
A list of allowable run states (*init(1)*) for package installation (for example, "*S s 1*").

RSTATES
A list of allowable run states (*init(1)*) for package removal (for example, "*S s 1*").

BASEDIR
The pathname to a default directory where relocatable files may be installed. If blank, the package is not relocatable and any files that have relative pathnames will not be installed. An administrator can override the default directory.

ULIMIT
If set, this parameter is passed as an argument to the *ulimit* command, which establishes the maximum size of a file during installation.

ORDER
A list of classes defining the order in which they should be put on the medium. Used by *pkgmk* in creating the package. Classes not defined in this field are placed on the medium using the standard ordering procedures.

MAXINST
The maximum number of package instances that should be allowed on a machine at the same time. By default, only one instance of a package is allowed. This parameter must be set in order to have multiple instances of a package.

PSTAMP
Production stamp used to mark the *pkgmap* file on the output volumes. Provides a means for distinguishing between production copies of a version if more than one is in use at a time. If *PSTAMP* is not defined, the default is used. The default consists of the Reliant UNIX system machine name followed by the string "*YYMMDDHHMM*" (year, month, date, hour, minutes).

INTONLY
Indicates that the package should only be installed interactively when set to any non-NULL value.

PREDEPEND
Used to maintain compatibility with pre-V5.41 package dependency chekking. Pre-V5.41 dependency checks were based on whether or not the name file for the required package existed in the */var/options* directory. This directory is not maintained for V5.41 packages since the *depend* file is used for checking dependencies. However, entries can be created in this directory to maintain compatibility. Setting the *PREDEPEND* parameter to *y* or *yes* creates a */usr/option* entry for the package. (Packages that are new for V5.41 do not need to use this parameter.)

EXAMPLES

Here is a sample *pkginfo* file:

```
PKG="oam"
NAME="OAM Installation Utilities"
VERSION="3"
VENDOR="AT&T"
HOTLINE="1-800-ATT-BUGS"
EMAIL="attunix!olsen"
VSTOCK="0122c3f5566"
CATEGORY="system.essential"
ISTATES="S 2"
RSTATES="S 2"
```

NOTES

Developers may define their own installation parameters by adding a definition to this file. A developer-defined parameter must begin with a capital letter.

NAME

pkgmap - package contents description file

DESCRIPTION

pkgmap is an *ASCII* file that provides a complete listing of the package contents. It is automatically generated by *pkgmk(1M)* using the information in the *prototype* file, and it should not be modified.

Each entry in *pkgmap* describes a single 'deliverable object file'. A deliverable object file includes shell scripts, executable objects, data files, directories, etc. The entry consists of several fields of information, each field separated by a space.

The fields are described below and must appear in the order shown.

part

An optional field designating the part number in which the object resides. A *part* is a set of objects, and is the atomic unit by which a package is processed. A developer can choose the criteria for grouping files into a part (e.g., based on class). If no value is defined in this field, *part=1* is assumed.

ftype

A one-character field that indicates the file type. Valid values are:

f - a standard executable or data file

e - a file to be edited upon installation or removal

v - volatile file (one whose contents are expected to change)

d - directory

x - an exclusive directory

l - linked file

p - named pipe

c - character special device

b - block special device

i - installation script or information file

s - symbolic link

class

The installation class to which the file belongs. This name must contain only alphanumeric characters and be no longer than 12 characters. It is not specified if the *ftype* is *i* (information file).

pathname

The pathname where the object will reside on the target machine, such as */usr/bin/mail*. Relative

pathnames (those that do not begin with a slash) indicate that the file is relocatable.

For linked files (*ftype* is either *l* or *s*), *pathname* must be in the form of *path1=path2*, with *path1* specifying the destination of the link and *path2* specifying the source of the link.

pathname may contain variables which support relocation of the file. A *$parameter* may be embedded in the pathname structure. *$BASEDIR* can be used to identify the parent directories of the path hierarchy, making the entire package easily relocatable. Default values for *parameter* and *BASEDIR* must be supplied in the *pkginfo* file and may be overridden at installation.

major

The major device number. The field is only specified for block or character special devices.

minor

The minor device number. The field is only specified for block or character special devices.

mode

The octal mode of the file (for example, 0664). A question mark (?) indicates that the mode will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked files, packaging information files or non-installable files.

owner

The owner of the file (for example, *bin* or *root*). The field is limited to 14 characters in length. A question mark (?) indicates that the owner will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked files or non-installable files. It is used optionally with a package information file. If used, it indicates with what owner an installation script will be executed.

Can be a variable specification in the form of $[A-Z]*. Will be resolved at installation time.

group

The group to which the file belongs (for example, *bin* or *sys*). The field is limited to 14 characters in length. A question mark (?) indicates that the group will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked files or non-installable files. It is used optionally with a package information file. If used, it indicates with what group an installation script will be executed.

Can be a variable assignment in the form of $[A-Z]*. Will be resolved at installation time.

size

The actual size of the file in bytes. This field is not specified for named pipes, special devices, directories or linked files.

cksum

The checksum of the file contents. This field is not specified for named pipes, special devices, directories or linked files.

modtime

The time of last modification, as reported by the *stat(2)* function call. This field is not specified for named pipes, special devices, directories or linked files.

Each *pkgmap* must have one line that provides information about the number and maximum size (in 512-byte blocks) of parts that make up the package. This line is in the following format:

:number_of_parts□□maximum_part_size

Lines that begin with "#" are comment lines and are ignored.

When files are saved during installation before they are overwritten, they are normally just copied to a temporary pathname. However, for files whose mode includes execute permission (but which are not editable), the existing version is linked to a temporary pathname and the original file is removed. This allows processes which are executing during installation to be overwritten.

EXAMPLES

The following is an example of a *pkgmap* file.

:2 500

```
1 i pkginfo 237 1179 541296672
1 b class1 /dev/diskette 17 134 0644 root other
1 c class1 /dev/rdiskette 17 134 0644 root other
1 d none bin 0755 root bin
1 f none bin/INSTALL 0755 root bin 11103 17954 541295535
1 f none bin/REMOVE 0755 root bin 3214 50237 541295541
1 l none bin/UNINSTALL=bin/REMOVE
1 f none bin/cmda 0755 root bin 3580 60325 541295567
1 f none bin/cmdb 0755 root bin 49107 51255 541438368
1 f class1 bin/cmdc 0755 root bin 45599 26048 541295599
1 f class1 bin/cmdd 0755 root bin 4648 8473 541461238
1 f none bin/cmde 0755 root bin 40501 1264 541295622
1 f class2 bin/cmdf 0755 root bin 2345 35889 541295574
1 f none bin/cmdg 0755 root bin 41185 47653 541461242
2 d class2 data 0755 root bin
2 p class1 data/apipe 0755 root other
2 d none log 0755 root bin
2 v none log/logfile 0755 root bin 41815 47563 541461333
2 d none save 0755 root bin
2 d none spool 0755 root bin
2 d none tmp 0755 root bin
```

NOTES

The *pkgmap* file may contain only one entry per unique pathname.

NAME

prototype - package information file

DESCRIPTION

prototype is an *ASCII* file used to specify package information. Each entry in the file describes a single deliverable object. An object may be a data file, directory, source file, executable object, etc. prototype is used by the pkgmk command in building the software package. This file is generated by the package developer.

Entries in a *prototype* file consist of several fields of information separated by white space. Comment lines begin with a "#" and are ignored. The fields are described below and must appear in the order shown.

part

An optional field designating the part number in which the object resides. A *part* is a set of objects, and is the atomic unit by which a package is processed. A developer can choose criteria for groupig files into a part (e.g., based on class). If this field is not used, *part=1* is assumed.

ftype

A one-character field which indicates the file type. Valid values are:

f - a standard executable or data file

e - a file to be edited upon installation or removal

v - volatile file (one whose contents are expected to change)

d - directory

x - an exclusive directory

l - linked file

p - named pipe

c - character special device

b - block special device

i - installation script or information file

s - symbolic link

class

The installation class to which the file belongs. This name must contain only alphanumeric characters and be no longer than 12 characters. The field is not specified for installation scripts. (*admin* and all classes beginning with capital letters are reserved class names.)

pathname

The pathname where the file will reside on the target machine, e.g., */usr/bin/mail* or *bin/ras_proc*. Relative pathnames (those that do not begin with a slash) indicate that the file will be installed relative to the package's default directory (*BASEDIR* in *pkginfo(4)*).

The form

path1=path2

may be used for two purposes: to define a link and to define local pathnames.

For linked files, *path1* indicates the destination of the link and *path2* indicates the source file. (This format is mandatory for linked files.)

For local pathnames, *path1* indicates the pathname an object should have on the machine where the entry is to be installed and *path2* is the relative or absolute pathname of the obect on the local machine where the package is being developed.

A pathname may contain a variable specification, which will be resolved at the time of installation. This specification should have the form $[A-Z]*.

major

The major device number. The field is only specified for block or character special devices.

minor

The minor device number. The field is only specified for block or character special devices.

mode

The octal mode of the file (for example, 0664). A question mark (?) indicates that the mode will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked files or packaging information files.

owner

The owner of the file (for example, *bin* or *root*). The field is limited to 14 characters in length. A question mark (?) indicates that the owner will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked files or packaging information files.

Can be a variable specification in the form of $[A-Z]*. Will be resolved at installation time.

group

The group to which the file belongs (for example, *bin* or *sys*). The field is limited to 14 characters in length. A question mark (?) indicates that the group will be left unchanged, implying that the file already exists on the target machine. This field is not used for linked files or packaging information files.

Can be a variable specification in the form of $[A-Z]*. Will be resolved at installation time.

An exclamation point (*!*) at the beginning of a line indicates that the line contains a command. These commands are used to incorporate files in other directories, to locate objects on a host machine, and to set permanent defaults. The following commands are available:

search

Specifies a list of directories (separated by white space) to search for when looking for file contents on the host machine. The basename of the *path* field is appended to each directory in the ordered list until the file is located.

include

Specifies a pathname which points to another *prototype* file to include.

default

Specifies a list of attributes (mode, owner, and group) to be used by default if attribute information is not provided for *prototype* entries which require the information. The defaults do not apply to entries in include *prototype* files.

param=value

Defines the indicated variable in the current environment.

The commands apply to the current *prototype* file only. They are not inherited by *prototype* files included by *!include* commands.

The above commands may have variable substitutions embedded within them, as demonstrated in the two example *prototype* files below.

Before files are overwritten during installation, they are copied to a temporary pathname. The exception to this rule is files whose mode includes execute permission, unless the file is editable (i.e, *ftype* is *e*). For files which meet this exception, the existing version is linked to a temporary pathname, and the original file is removed. This allows processes which are executing during installation to be overwritten.

EXAMPLES

Example 1 of prototype(4)

```
!PROJDIR=/usr/proj
!BIN=$PROJDIR/bin
!CFG=$PROJDIR/cfg
!LIB=$PROJDIR/lib
!HDRS=$PROJDIR/hdrs
!search /usr/myname/usr/bin /usr/myname/src /usr/myname/hdrs
i pkginfo=/usr/myname/wrap/pkginfo
i depend=/usr/myname/wrap/depend
i version=/usr/myname/wrap/version
d none /usr/wrap 0755 root bin
d none /usr/wrap/bin 0755 root bin
! search $BIN
f none /usr/wrap/bin/INSTALL 0755 root bin
f none /usr/wrap/bin/REMOVE 0755 root bin
f none /usr/wrap/bin/addpkg 0755 root bin
!default 755 root bin
f none /usr/wrap/bin/audit
f none /usr/wrap/bin/listpkg
f none /usr/wrap/bin/pkgmk
```

Example 2 of prototype(4)

```
# the following file starts out zero length but grows
v none /usr/wrap/logfile=/dev/null 0644 root bin
# the following line specifies a link
l none /usr/wrap/src/addpkg=/usr/wrap/bin/rmpkg
! search $SRC
!default 644 root other
f src /usr/wrap/src/INSTALL.sh
f src /usr/wrap/src/REMOVE.sh
f src /usr/wrap/src/addpkg.c
f src /usr/wrap/src/audit.c
f src /usr/wrap/src/listpkg.c
f src /usr/wrap/src/pkgmk.c
d none /usr/wrap/data 0755 root bin
d none /usr/wrap/save 0755 root bin
d none /usr/wrap/spool 0755 root bin
d none /usr/wrap/tmp 0755 root bin
d src /usr/wrap/src 0755 root bin
```

Example 3 of prototype(4)

```
# this prototype is generated by 'pkgproto' to refer
# to all prototypes in my src directory
!PROJDIR=/usr/dew/projx
!include $PROJDIR/src/cmd/prototype
!include $PROJDIR/src/cmd/audmerg/protofile
!include $PROJDIR/src/lib/proto
```

NOTES

If a non-existing file is defined in the prototype file, this file is usually created when the package is installed. If the pathname of the file, however, contains a non-existing directory, the file is not created. If, for example, the prototype file contains the following entry:

f none /usr/dev/bin/command

and the file does not exist, it is only created if the /usr/dev/bin directory already exists or if the prototype file also contains an entry defining the directory:

d none /usr/dev/bin

SEE ALSO

pkginfo(4), pkgmk(1M).

NAME

space - disk space requirement file

DESCRIPTION

space is an *ASCII* file that gives information about additional disk space requirements for a software package. It defines space needed beyond that which is used by objects defined in the *prototype* file - for example, files which will be installed with the *installf* command. It should define the maximum amount of additional space which a package will require.

The generic format of a line in this file is:

pathname□□blocks□□inodes

Definitions for the fields are as follows:

pathname
: Specifies a directory name which may or may not be the mount point for a filesystem. Names that do not begin with a slash (/) indicate relocatable directories (see the *BASEDIR* variable in *pkginfo(4)*).

blocks
: Defines the number of additional disk blocks required for installation of the files and directory entries contained in the *pathname* directory (using a 512-byte block size).

inodes
: Defines the number of additional inodes required for installation of the files and directory entries contained in the *pathname* directory.

EXAMPLE

Example of space(4):

```
#□extra□space□required□by□config□data□which□is
#□dynamically□loaded□onto□the□system
data□□500□□□1
#□extra□space□required□under□/opt
/opt□□200□□15
```

SEE ALSO

installf(1M), prototype(4), pkginfo(4)

Related publications

Ordering manuals

The manuals can be ordered from your local Siemens Nixdorf office.Some of the more frequently required manuals are also available online. The online documentation is available on CD and can also be accessed on the World Wide Web from Siemens Nixdorf's homepage.

[1] **Reliant UNIX**
Operating Manual (hardware-specific)

Target Group
System administrators, users

Contents
This manual describes how the hardware is installed, and how the system is placed in service and operated. The contents of the manual are dependent on the computer type.

[2] **Reliant UNIX 5.44**
Reliant UNIX Installation and Operation, RM200, RM300, RM400
Installation Guide

Target Group
System administrators

Contents
This manual describes how to install the Reliant UNIX 5.44 operating system both as a new installation and as an update. Also documented are the basics of the operating system and key aspects of using it as well as instructions for installing the software using the SYSADM operator system.

[3] **SINIX V5.40**
ANSI C and Programming Support Tools
Programmer's Guide

Target Group
Programmers

Contents
This book describes UNIX system tools supplied with the C compilation system. It concentrates on the compilation system and the program development tools lint, sdb, lprof, and cscope, make and SCCS, lex, yacc, and m4.

[4] **Reliant UNIX 5.44**
Commands, User Reference Manual
Reference Manual

Target Group
Users, system administrators

Contents
This manual is intended for reference purposes. It describes the user commands of the Reliant UNIX 5.44 operating system.

[5] **Reliant UNIX 5.44**
Reliant UNIX Installation and Operation, RM600-xxx, RM600-E
Installation Guide

Target Group
System administrator

Contents
This manual describes how to install the Reliant UNIX 5.44 operating system both as a new installation and as an update. Also documented are the basics of the operating system and key aspects of using it as well as instructions for installing the software using the SYSADM operator system.

[6] **Reliant UNIX 5.44**

Documentation Guide
Annotated Overview, Index

Target Group
Users and system administrators, network users and administrators, Spool users and administrators and programmers.

Contents
The "Documentation Guide" provides an overview of the documentation available for Reliant UNIX as well as the titles of the manuals and their order numbers. As well as a listing of the contents of the most important manuals, an index is provided which sets out the activities covered by the respective manuals and acts as a reference to the manual where the actual activity is documented.

[7] **Reliant UNIX 5.44**
System Administrator's Reference Manual
Reference Manual

Target Group
System administrators

Contents
This manual describes the system administrator commands, file formats, device files and procedures for system maintenance in alphabetical order.

[8] **Reliant UNIX 5.44**
System Administrator's Guide
User Guide

Target Group
System administrators

Contents
This manual provides a basic overview of Reliant UNIX system administration, with information about important system features and the most critical tasks for the system administrator.

[9] **SINIX V5.40**
CES C Development System, Part 1
User Guide

Target Group
C programmers

Contents
The C compiler C compiler messages Overview of the ANSI C language elements

[10] **SINIX V5.40**
CES C Development System
User Guide

Target Group
C programmers

Contents
This manual discusses the most important tools for the development, management, maintenance, and generation of C programs, and also explains the format of object files.

[11] **Reliant UNIX 5.44**
User's Guide
User Guide

Target Group
Users

Contents
The "Users Guide" describes the main elements of the Reliant UNIX operating system, including an introduction to using SINIX, the file system, process handling and the shell.

[12] **Reliant UNIX 5.44**
Network Administration
System Administrator's Guide

Target Group
Network administrators managing UNIX systems in TCP/IP networks

Contents
This manual describes the network administration activities that have to be performed when using the TCP/IP software on Reliant UNIX 5.44 as well as the basic network function (BNU).

[13] **Reliant UNIX 5.44**
Network Reference Manual
Reference Manual

Target Group
Network administrators, system administrators

Contents
This manual gives a description in alphabetical order of the commands, C functions, file formats, system maintenance procedures as well as other Reliant UNIX tools for operating networks.

[14] **SINIX/windows V3.1**
Documentation Overview

Target Group
All users

Contents
This edition of the Documentation Overview applies to the documentation for SINIX V5.42 and above on the following systems: RM1000, RM600, RM400, RM200. The English documentation for this software consists of 28 books and is so varied and wide-ranging that even experienced users faced with specific difficulties do not always know which manual contains the information they need. The overview aims to address that problem.

[15] **SINIX V5.40**
Character User Interface
Programmer's Guide

Target Group
Programmers

Contents
Description of the tools required to build interfaces for communicating with non-graphics terminals.

[16] **SINIX**
Device Driver Interface/Driver Kernel Interface
Reference Manual

Target Group
System programmers

Contents
Creation, modification and integration of drivers which run under UNIX System V Release 4

[17] **SINIX V5.41**
STREAMS
Programmer's Guide

Target Group
System developers, application programmers designing driver and communication software

Contents
The manual describes the UNIX System V STREAMS mechanism. It provides a general overview, deals with the individual components and supplies programming guidelines. It also includes a reference section.

[18] **Reliant UNIX 5.44**
System Administration and Hardware Configuration Using the SYSADM User Interface
System Administrator's Guide

Target Group
System administrators

Contents
This manual describes the SYSADM user interface with the integrated configuration tool Config. The Config is available under the menu option Configuration once the operating system has been installed in SYSADM. The user interface provides a user-friendly means of configuring and managing I/O devices.

[19] **Reliant UNIX 5.44**
Hardware Configuration with Config under SINIX/windows

Target Group
System administrators

Contents
This manual describes the functionality of Config under the SINIX/windows user interface.

[20] **Reliant UNIX 5.44**
Virtual Disks
User Guide

Target Group
System administrators

Contents
This manual describes the various types of virtual disk and how they are configured.

[21] **Reliant UNIX 5.44**
Veritas File System (VxFS) V1.2
System Administrator's Guide

Target Group
System administrators

Contents
This manual provides an overview of the structure and the features of the Veritas file system (VxFS) and also describes how it is used. In addition, the commands and utilities used with VxFS are documented and detailed explanations of error and diagnostic messages are given.

[22] **Reliant UNIX 5.44**
Tuning Guide
System Administrator's Guide

Target Group
System administrators

Contents
This manual describes options for improving the system performance.

[23] **Xprint V5.0**
Reference Manual

Target Group
Reliant UNIX users and system administrators; Spool administrators

Contents
This manual describes the Spool commands in alphabetical order, and documents the Spool messages, the configuration files for Spool objects as well as the standard Spool data formats. The manual also presents an overview of how the Spool system works.

[24] **SINIX**
AT&T Spool

System Administrator's Guide

Target Group
System administrators

Contents
This manual describes how to set up and manage the LP print service.

[25] **SINIX/windows User Environment V3.1**
Documentation Overview

Target Group
SINIX/windows users

Contents
The manual provides an overview of the SINIX/Windows documentation.

[26] **SINIX/windows User Environment V3.0 (CDE)**
User's Guide
User Guide

Target Group
Users with basic knowledge of Reliant UNIX

Contents
This manual describes the underlying features of the SINIX/windows desktop and explains how to work with the desktop and the desktop applications.

[27] **SINIX/windows User Environment V3.0 (SINIX Desktop)**
Introduction to Handling and Configuration
User Guide

Target Group
Users with basic knowledge of Reliant UNIX

Contents
This manual provides an introduction to operating and configuring the SINIX/windows graphical user interface. It describes the components of the SINIX/windows desktop, how to use the user interface, and interactive configuration of the desktop.

[28] **SINIX/windows User Environment V3.0 (CDE)**
Advanced User and System Administrator Guide
System Administrator's Guide

Target Group
System administrators

Contents
This manual covers advanced tasks in customizing the appearance and behavior of the Desktop.

[29] **SINIX/windows User Environment V3.1 (CDE)**
CDE Enhancements
User Guide

Target Group
Users and system administrators

Contents
The book provides an overview of the additional functionality that Siemens Nixdorf Informationssysteme AG an Triteal Corporation have added to the Common Desktop Environment (CDE).

[30] **SINIX/windows User Environment V3.0 (SINIX Desktop)**
Guide for Experts and System Administrators
System Administrator's Guide

Target Group
System administrators

Contents
This manual discusses the concepts underlying SINIX/windows, such as the client-server model, and configuration of a client using resources. It also describes the primary clients.

[31] **SINIX/windows User Environment**
Clients Reference Manual
Reference Manual

Target Group
System administrators

Contents
This manual provides the experienced user with a comprehensive overview of the SINIX/windows clients and the resources used by the clients.

[32] **SINIX/windows User Environment**
XDCL Desktop Configuration Language
Reference Manual

Target Group
System administrators

Contents
This manual provides an overview of the XDCL language,with which the experience user can configure the SINIX/windows desktop.

[33] **SINIX**
UNIX System Security
User Guide

Target Group
Users, system administrators, system programmers

Contents
This manual describes the user classes of a UNIX system, provides an overview of the security risks in a UNIX system, and explains the ways of reducing these risks.

[34] **SINIX**
High Availability Guide
User Guide

Target Group
System administrators, Systemsicherheits-Beauftragte

Contents
The High Availability Guide describes ways of increaing the availability of UNIX systems.

[35] **SINIX**
OBSERVE
User Guide

Target Group
OBSERVE system administrators and OBSERVE programmers

Contents
This manual describes how OBSERVE is installed and configured in single-, dual- and multi-computer configurations.

[36] **Reliant UNIX**
AUDIT V2.0
System Administrator Guide

Target Group
System administrators

Contents
This manual describes how AUDIT is installed and configured.

[37] **Reliant UNIX 5.44**
UFS UNIX File System
System administrators

Target Group
System administrators

Contents
This manual describes the organization, administration, maintenance and consistency checking for the ufs file system.

[38] **Reliant UNIX 5.44**
Error Diagnosis and Error Handling

Target Group
System administrators, programmers and service technicians

Contents
This manual describes potential system errors, how these errors are interpreted, as well as possible measures for dealing with errors. In addition, the user interfaces available for system diagnostics are described.

[39] **IOCS V3.0**
Administration, Configuration and Programming of Printers

Target Group
System administrators, application programmers

Contents
This manual describes how to manage, configure, install and program printers using IOCS and also outlines the compatible control sequences for printer programming. It provides information on programming with the IOCS but does not include programming information.

[40] **Reliant UNIX, Windows NT**
RAIDmaster
System Administrator's Guide

Target Group
System administrators

Contents
The manual describes the configuration of the RAIDmaster-controllers.

[41] **RMS V3.1**
RMS Installation

Target Group
System administrators

Contents
This manual describes how to install the Reliant Monitor Software (RMS).

[42] **RMS V3.1**
RMS Operation

Target Group
System administrators

Contents
This manual describes the Reliant Cluster Visual Manager (RCVM), the graphical user interface for RMS.

[43] **RMS V3.1**
RMS Configuration and Administration

Target Group
System administrators

Contents
This manual decribes the configuration and administration of RMS. The manual also presents an overview of RMS, gives examples how to configure RMS and how to administer RMS via the command interface.

[44] **SIDLM V1.2**
Installation and Administration Guide

Target Group
System administrators

Contents
The manual decribes how to install the SINIX Distributed Lock Manager (SIDLM) . Also documented is the administration of a SIDLM cluster.