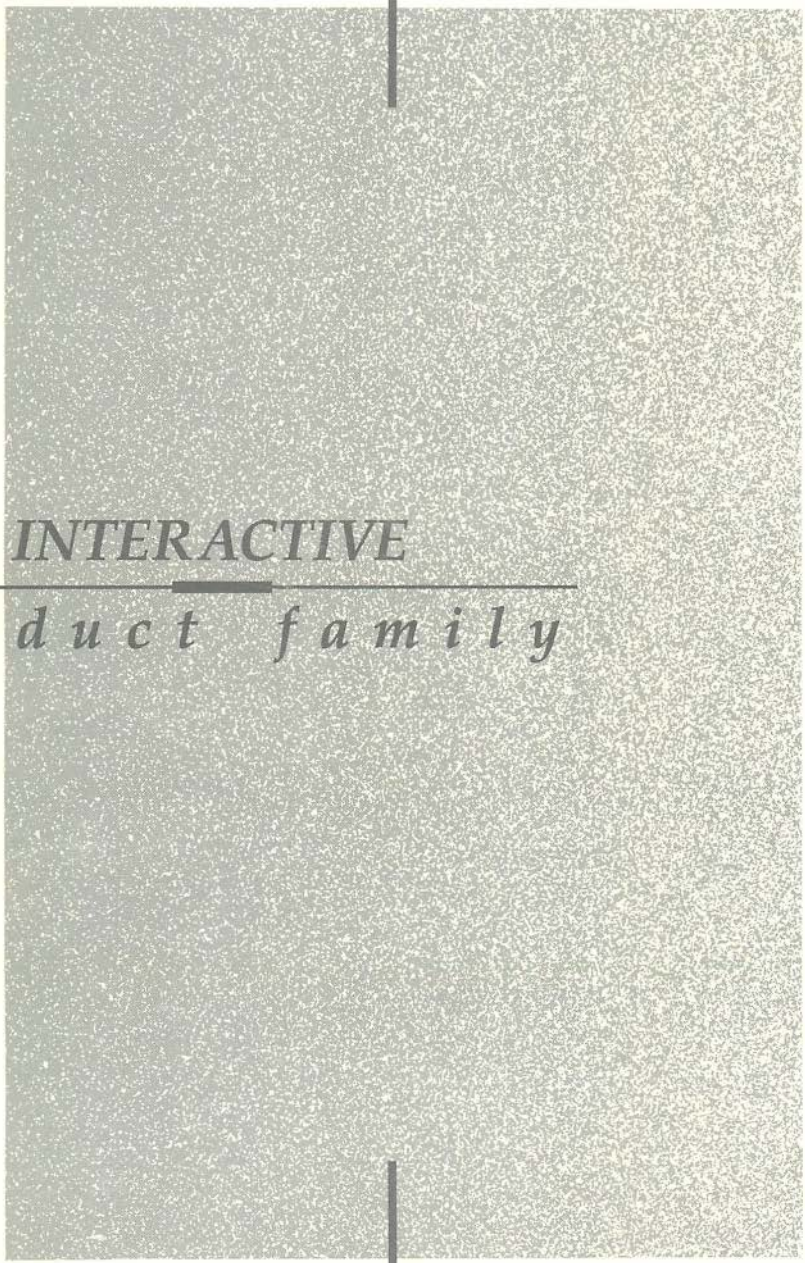
UNIX* System V/386 Release 3.2 Integrated Software Development Guide



SunSoft
A Sun Microsystems, Inc. Business

UNIX® System V/386 Release 3.2

Integrated Software Development Guide

If you plan to write applications or drivers that will be integrated with the INTERACTIVE UNIX Operating System, please be sure to read "Integrating Software With the INTERACTIVE UNIX Operating System" in the *INTERACTIVE Software Development System Guide and Programmer's Reference Manual,* which supplements the information found in this document.

Note that some of the manual pages in Appendix A are superseded by those in the *INTERACTIVE UNIX System V/386 Release 3.2 User's/System Administrator's Reference Manual* and the *INTERACTIVE Software Development System Guide and Programmer's Reference Manual.*

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

Table of Contents

1 Introduction

Introduction

Purpose of This Guide

The *Integrated Software Development Guide* (ISDG) is intended for the Independent Software Vendor (ISV) who develops UNIX System software applications to run on 386 computer systems. The *Guide* supplies the information needed to write application software and installable drivers for new hardware additions for UNIX System V/386 Release 3.2 (Version 1.0). Also included is information on the use of the keyboard, the screen, remote terminals, and supported printer subsystems.

Adherence to these guidelines helps to ensure compatibility both with the current 386 processor and with its future enhancements.

What Is Covered In This Guide

The material in this *Guide* is organized into the following chapters:

- Chapter 1, *Introduction*, briefly describes what is included in this *Guide* and how to use it.
- Chapter 2, *UNIX Application Software Installation*, outlines the procedure you use to install the UNIX System software and provides the details necessary to create a software installation floppy disk set for your computer. Some broad guidelines are also presented for installing and removing UNIX programs, as well as examples of installing and removing scripts.
- Chapter 3, *Device Drivers*, contains the rules and procedures you need to follow for writing device drivers for the UNIX System V/386. As you may know, writing a device driver carries a lot of responsibility because, as part of the UNIX Operating System kernel, it is assumed to always take the correct action.
- Chapter 4, *Porting*, discusses recommended approaches to programming and preferred programming techniques. As such, it presents porting considerations only and is not an exhaustive reference about porting.

- Chapter 5, *Security*, describes enhancements to security made in this release of UNIX System V/386.
- Appendix A, *Manual Pages*, contains the manual pages for those ID programs and files a writer of device drivers needs to know about.
- Appendix B, *A Simple Game Port Driver*, discusses game controller hardware as a basis for an elementary UNIX device driver.
- Appendix C, *The Trace Driver*, presents a pseudo-device, called the "trace driver," that allows the UNIX Operating System kernel or other device drivers to report debugging information without the use of console printf's.
- Appendix D, *A Prototype Floppy Disk Driver*, contains some selected portions of the UNIX System V/386 floppy disk device driver source files.
- Appendix E, *A Sample Driver Software Package*, shows the ID modules needed to install a device driver and describes the **Install** and **Remove** scripts.
- Appendix F, *Porting*, lists the utility programs required on UNIX System V/386 and which can be used in portable programs.

An index is included at the end of the *Guide*.

How to Use This Guide

To use this *Guide* effectively, you should already have a good working knowledge of the UNIX Operating System. This *Guide* is intended as a reference to help you in designing device drivers and other software development activities.

Notational Conventions

The following notational conventions are used throughout this *Guide*:

bold	User input, such as commands, options to commands, and names of directories and files, appear in **bold**.
italic	Names of variables to which values must be assigned (such as *filename*) appear in *italic*.
`constant width`	UNIX System output, such as prompt signs and responses to commands, and programming examples appear in `constant width`.
<>	Input that does not appear on the screen when typed, such as passwords, keys used as commands, or RETURN and other special keys, appear between angle brackets.
<^*char*>	Control characters are shown between angle brackets because they do not appear on the screen when typed. The circumflex (^) represents the control key (usually labeled CTRL). To type a control character, hold down the control key while you type the character specified by *char*. For example, the notation <^**d**> means to hold down the control key while pressing the D key; the letter D will not appear on the screen.
[ ]	Command options and arguments that are optional, such as **[-msCj]**, are enclosed in square brackets.
¦	The vertical bar separates optional arguments from which you may choose one. For example, when a command line has the format *command* [*arg1* ¦ *arg2*] you may use either *arg1* or *arg2* when you issue *command*.

...	An ellipsis after an argument means that more than one argument may be used on a single command line. A vertical ellipsis is used in programming examples to indicate missing portions of code.
command(number)	A command name followed by a number in parentheses refers to the part of a UNIX System reference manual that documents that command. (There are two reference manuals: the *User's/System Administrator's Reference Manual* and the *Programmer's Reference Manual*.) For example, the notation *cat*(1) refers to the page in section 1 of the *User's/System Administrator's Reference Manual* that documents the **cat** command.

Related Documentation

A variety of documents support the UNIX System V/386. Refer to the *Documentation Roadmap*, which helps you get acquainted with the documents you can use with UNIX System V/386 Release 3.2.

The *Roadmap* helps you to understand general relationships among the documents and to identify which documents you want to order.

Throughout this *Guide*, references are made to certain specific documents listed in the *Documentation Roadmap*. Rather than list the complete title each time the document is referenced, the following convention is used:

- The *UNIX System V/386 Release 3.2 Programmer's Guide* is referred to as the *Programmer's Guide*.
- The *UNIX System V/386 Release 3.2 Programmer's Reference Manual* is referred to as the *Programmer's Reference Manual*.
- The *UNIX System V/386 Release 3.2 User's/System Administrator's Reference Manual* is referred to as the *User's/System Administrator's Reference Manual*.

- The *UNIX System V/386, Release 3, Block and Character Interface, Device Driver Reference Manual*, Select Code 307-192, September 1987, is referred to as the *Device Driver Reference Manual*.

2 Unix Application Software Installation

UNIX Application Software Installation

This chapter outlines the procedure used to install UNIX System add-on software and provides the developer with the details necessary to create a software installation floppy disk set for UNIX System V/386. Some broad guidelines are also provided for the installation and removal of programs, along with examples of the installation and removal programs.

Do not use commands that do not reside on the base system. It is strongly recommended that the **Install** and **Remove** programs be tested on the base system alone to ensure their proper execution. If your application uses commands that reside in other foundation set packages (apart from the base system), your documentation must explicitly tell the user which packages must be installed for it to work correctly. Refer to the section "Commands Not Part of the Base System" in this chapter for more information.

Installation Scenario

Installation of software on UNIX System V/386 is done with the **installpkg** procedure that is available from the UNIX System command prompt.

The user is instructed to insert the software floppy disks in order, as prompted by the system until all of the disks have been read.

Installation Tools

cpio

The basic mechanism to transfer software from a floppy disk to the UNIX System V/386 hard disk is **cpio**. The most common forms of the **cpio** command are as follows:

To create a floppy:	**cpio -ocB > /dev/rdsk/f0d9d** (360KB)
	cpio -ocB > /dev/rdsk/f0q15d (1.2MB)
	cpio -ocB > /dev/rdsk/f0 (1.44MB, other)
To read from a floppy:	**cpio -icBd < /dev/rdsk/f0d9d** (360KB)
	cpio -icBd < /dev/rdsk/f0q15d (1.2MB)
	cpio -icBd < /dev/rdsk/fo (1.44MB, other)

The **cpio** command does not format floppies. Floppy disks must be formatted. See the *cpio*(1) manual pages in the *User's/System Administrator's Reference Manual* for a full discussion of the **cpio** command and its parameters. For information on formatting, see the *format*(1M) manual page in the *User's/System Administrator's Reference Manual.*

Special Installation Files

The key to the software installation procedure is information stored in the special installation files. These special installation files must be included by the developer on the floppies together with the actual software files. It is the existence of these files that provides the smooth and friendly user-interface and some error protection. Naturally, these files must also appear on the **cpio** list. These files are the following:

1. **Size**: This is an ASCII file that contains information about the size of the floppy set. The complete format is presented later in the section "The Size File."
2. **Install**: This is an executable file or shell script. It is executed *after* the files are copied (by means of **cpio**) from the floppy disk into the temporary area and *before* they are removed from the temporary installation directory. The contents of **Install** is the responsibility of the software developer. See the section "The Install Program" for a discussion of how to create an **Install** file.
3. **Name**: This file contains descriptive information about the application being installed (information that can be made available to the user at installation time). The format of **Name** is described in the section "The Name File."
4. **Remove**: This is an executable file or shell script. The function is to remove the software package. See the section "Removal of Installed Software" for a discussion of the **Remove** file.

5. **Files**: This file contains the full pathnames of the files that are being installed. **Files** must list the absolute pathname for each file contained on the software installation floppy or created by the **Install** program. The names must be completely enumerated and contain no wildcard characters (for example, asterisks, question marks, etc.).

Installation Program

The **installpkg** command works as follows:

1. **Prompt the User for the First Floppy**: The user, presented with a message, is asked to insert the first floppy disk and strike <ENTER> when ready.
2. **Check Available Disk Space**: From the first installation floppy disk, the **Size** file is read, and the size of the software to be installed is ascertained. If there is sufficient space left on the file systems to store the new software, the procedure continues. If not, it aborts with an appropriate error message.

3. **Copy the Floppy Disk Into a Temporary Directory**: A temporary directory is created. (Note that this directory will be removed when the installation is complete.) The entire floppy disk set is copied into the temporary directory by means of the **cpio -icBd < /dev/rdsk/f0...** command. It is important to note that for the **cpio** command, the pathnames on the software installation floppy disk **must be relative (must not begin with a /)** so that the existing files are not accidentally corrupted at this stage. Note that applications may be installed transparently from either 360 KB, 1.2 MB, or 1.44 MB floppy disks.

   Note that the temporary directory will be **/usr/tmp/install***xxxx* where *xxxx* is the process id (PID) of the install process, but the **Install** script should not assume this.

4. **Check Installation Files for Completeness**: The files copied into the temporary directory are checked to see if they include all of the special files. They are as follows:

   ./Size
   ./Install
   ./Name
   ./Remove
   ./Files (optional)

   The specific formats for these files are specified in the section titled "Creation of the Software Installation Floppy Disk Set."

5. **Execute Install (./Install) Relocate Application Programs**: The vendor-supplied install program (stored by **cpio** in **./Install**) is executed. This program, written by the software vendor, must move the application out of the temporary directory and install any drivers. Refer to Chapter 3 on writing device drivers. Instructions on how to create **./Install** can be found in the section titled "The Install File." If **./Install** is a success (that is, returns a value of zero), the installation proceeds to the next step. If **./Install** fails (that is, returns a nonzero value), the installation aborts.

It is the vendor's responsibility to remove any previously installed fragments of the aborted installation and present the user with a statement saying that the installation was aborted and why.

6. **Update System Files**: The **Name** file is renamed to a unique identifier and stored in **/usr/options**. The **Files** file (vendor-supplied) is renamed and located in **/usr/lib/installed/Files**. The **Remove** program (also vendor-supplied) is renamed and located in **/usr/lib/installed/Remove**.
7. **The Temporary Directory is Removed**: The temporary directory (including any files that remain within the directory) is removed.
8. **Success Message Issued**: A success message is issued, and control is returned to the UNIX System prompt.

Creation of the Software Installation Floppy Disk Set

Do not use commands that do not reside on the base system. It is strongly recommended that the **Install** and **Remove** programs be tested on the base system alone to ensure their proper execution. Refer to the section "Commands Not Part of the Base System" for more information.

Format of the Floppy Disks

The installation disks should be created with the **cpio** command. The possible argument options are

cpio -ocB > /dev/rdsk/f0q15d	(for 1.2MB format)
cpio -ocB > /dev/rdsk/f0d9d	(for 360KB format)
cpio -ocB > /dev/rdsk/f0	(for 1.44MB format)

The **cpio** command does not format floppies. Floppy disks must be formatted. See the *cpio*(1) manual pages in the *User's/System Administrator's Reference Manual* for a full discussion of the **cpio** command and its parameters. For information on formatting, see the *format*(1M) manual page in the *User's/System Administrator's Reference Manual*.

It is imperative that one of these specific options be used since the installation program that reads the floppy assumes this format.

The list of pathnames that drives the **cpio** option has two restrictions:

1. **All pathnames MUST be relative (NOT begin with a /).** This is because the **cpio** command will copy the files into the temporary directory before the **Install** program moves them into their permanent locations.
2. The first entry on the list should be **Size**. If the application requires more than one floppy, the **Size** file must be on the first floppy and should appear first.

Following this special file should be the pathnames (without a leading /) for all files to be placed on the floppy.

The floppy set should be clearly labeled with the **name of the product** (as it appears in the **Name** file), **the version of the product** (which may include the date), the **floppy number** (for example, number 2 of 5) showing the total number in the package, and the **disk format** (360 KB, 1.2 MB, or 1.44 MB).

Name File

The **Name** file contains the name of the product as it is to appear during the installation and removal operations. This file is a single record (or single line) and can contain up to 65 characters of significant data (only the first 65 characters are displayed in the menus). The **Name** file is used to determine whether the package has been previously installed. It should be unique. Avoid the use of shell metacharacters (for example, ^ / * ? [ ] & ! $) and single or double quotes in the **Name** file.

Size File

The **Size** file may be a single- or double-record file. It contains the number of blocks required on the root (/) and user (**/usr**) file systems to install and use the application package. A block is defined as 512 bytes.

Each number put in the **Size** file should be the larger of the two:

1. number of blocks that are used while installing the package; or
2. amount of blocks used up on the system once your package is installed and exercised by the user for each of the two file systems: root and user. Note that the root file system typically consists of all directories on the system, *except* those under the **/usr** directory.

To determine how much space the application requires on each file system, do the following for the application files residing in **/usr** and the remainder of the system:

Note that the following assumes 1.2MB.

1. Create a floppy set with **cpio -ocB > /dev/rdsk/f0** to contain the application files on each file system, one at a time.
2. Use **cd** to change to a temporary directory you created.

   If any files exist in this directory, move the files into another temporary directory or remove the files.
3. Type in **cpio -icBdu < /dev/rdsk/f0** for each floppy set created in Step 1. Do this one at a time.

   There should not be any files in the current directory prior to this **cpio** command.
4. Type **du -s .** to get the number of blocks. If the blocking format is not 512 bytes, perform a conversion.
5. Create the **Size** file as follows:

   USR=<number of blocks in /usr>
   ROOT=<number of blocks on />

   These lines can be in reverse order, and *no* spaces should be on either side of the equal sign.

Even if your system has only one file system, you must specify the requirement for both since some users may have both. If the user installing the package has only one file system, **installpkg** will add the two amounts you specify in the **Size** file to calculate the space requirement.

installpkg will make use of this file as follows:

1. The following message will be displayed:

```
Please insert the floppy disk.
```

```
If the program installation requires more than
one floppy disk, be sure to insert the disks in
the proper order, starting with disk number 1.
After the first floppy disk, instructions will
be provided for inserting the remaining floppy
disks.

Strike ENTER when ready
or ESC to stop.
```

2. When this is completed, the following in-progress message appears:

```
Installation is in progress-do not remove the floppy disk.
```

3. The **Size** file will be extracted from the first floppy.

   Caution should be exercised to prevent running out of disk space during software installation. There is no protection other than the **Size** file that can prevent this.

4. If the **Size** file does not exist, the package is presumed not to be valid. As long as **Size** is on the first floppy, lines of the form

```
USR=<amount>
ROOT=<amount>
```

   are read in, where <amount> is an integer number of 512-byte disk blocks.

Install File

The user should be notified when the **Install** program starts. The example

```
echo "Installing Fruit Commands"
```

is shown in the sample script "Install Fruit Example" in this chapter. If the **Install** program takes a long time to run, the user should be periodically informed of either what is happening or that the installation is still in progress and proceeding well. The user should not see certain system messages. For example, the user should be isolated from the **stderr** messages of commands that fail. (These should be picked up by the installation program and, if necessary, translated into meaningful user messages.) Similarly, when creating a directory or removing files, **stderr** and **stdout** should be redirected.

Transferral of Programs From the Temporary Directory

The **Install** program will not be invoked until all the programs and files on all floppies have been copied into the temporary directory. After that, the **Install** program will modify them as needed (for example, change mode or change ownership). The files will then be ready to be moved into their intended file system destination. The transfer (move) is usually a straightforward operation, but there are some common stumbling blocks.

Use the **mv** command to avoid possible disk space problems stemming from multiple copies of a single file.

- **Permissions:** It is advisable to ascertain that permissions are acceptable when created by the **cpio** command. Executables should have the correct executable permissions set, and read-write permissions should be such that the normal user can use them as required. The **setuid** bit should also be set where appropriate.
- **Ownership:** In general, files should be owned by standard system users (for example, **root, bin, uucp, lp, install**, etc.). Be careful not to include files that are owned by local users with specific machine ownership (such as the machine where the files were created).
- **Temporary Files:** If any intermediate files are created, place them in the temporary directory to ensure that fragments are not left. **installpkg** removes this directory at the end of the installation.

When **mv** is used to move application files from the temporary directory to the target directory, the file ownership and group attributes may change if the temporary directory and the target directory reside on different filesystems. When **mv** is used to move files across filesystems, the resulting file will be created with the owner and group attributes of the invoking user. Since **installpkg** is invoked as root, the resulting files will be created with root ownership. This may present application problems. Since it is not guaranteed that the temporary and target directories reside on the same filesystem, the installation script will force the correct owner and group attributes of the target files using **chown** and **chgrp**.

Installation of Libraries, Include Files, Etc.

The overriding concern here is "NOT TO OVERWRITE ANYTHING THAT ALREADY EXISTS." Do not redefine any libraries, include files, or **/etc/inittab**. Similar restrictions exist for any standard UNIX System file or executable.

Communication with the User

All communication with the user should be by means of the standard input and output (**stdin/stdout—echo/read**). It is generally advisable to keep the user informed as to the state of events, especially delays. When difficulties arise, alternative procedures can be presented and the user consulted. Ask the user for verification when necessary. The **installpkg** program issues its own completion message.

Abnormal Termination

If an installation procedure must be abnormally terminated (aborted), the program must return a nonzero exit status (use exit 1). This inhibits **installpkg** from

- giving the user the confirmation message that the package was installed
- updating the system files to show that the package has been installed

It is the responsibility of the **Install** program (and not **installpkg**) to remove the files already installed outside of the temporary directory. One simple way to accomplish this is to execute the **Remove** program. When an installation aborts, the user must be informed.

Reasons for abnormal termination include

- Required libraries may be missing.
- The user may have inadvertently removed commands that the **Install** program uses.
- The **Install** program may contain an error.

Removal of Installed Software

Do not use commands that do not reside on the base system. It is strongly recommended that the **Install** and **Remove** programs be tested on the base system alone to ensure their proper execution. Refer to the section "Commands Not Part of the Base System" for more information.

The UNIX System **removepkg** command takes as an argument the name of an installed package either from the **displaypkg** command or from the user who, when prompted, selects a name to be removed from a sorted list of the installed package names. Selection of the package invokes the **Remove** program that was originally supplied with the installation software set (currently residing in the **/usr/lib/installed/Remove** directory).

The quality and effectiveness of the **Remove** program is entirely dependent upon its author. The **removepkg** program checks neither for effectiveness nor completeness. The program looks only for an exit status. If the **Remove** program exits with an error or an abort, the existing software listings will NOT be changed. Only if the **Remove** program returns a success code will the software listings be adjusted to reflect the deletion.

As with the **Install** program, progress messages should be issued to the user, and system messages (unsolicited and otherwise) should be filtered and modified. **stderr** and **stdout** should be redirected (possibly to **/dev/null**).

Commands Not Part of the Base System

Although your **Install** and **Remove** scripts should not use commands not installed as part of the UNIX System Foundation Set Base System (see the *User's/System Administrator's Reference Manual* for more detail on what commands are installed as part of the Base System), your application may need to make use of them. If this is the case, you need to do two things:

1. Point out to the user in clear terms in your user documentation which UNIX System Foundation Set packages must be installed before the application may be used.
2. At the very beginning of your **Install** script, perform the following steps. After the list of steps is an example of how to implement them:

   - ☐ Determine whether or not the full complement of Foundation Set packages you need to run your application is installed. This can be done relatively easily by searching through the directory called **/usr/options** for the names of the packages. See example below.
   - ☐ If any of the packages are not installed, your **Install** script should issue the appropriate message and exit with a failure code immediately.

For example, suppose your application required commands from both the "Editing Package" and the "Network Support Utilities Package (1.1)" in addition to the Base System package. Both of these packages are included with the Foundation Set, but suppose the user had installed only the "Editing Package" in addition to the Base System. Your **Install** script would then need to execute the following code (or facsimile thereof) near the very beginning of the program:

```
...
e='grep "^Editing Package" /usr/options/* 2> /dev/null'
n='grep "^Network Support Utilities .*1.1" /usr/options/* 2> /dev/null'

if [ -z "$e" -o -z "$n" ]
   then
   if [ -z "$e" -a -z "$n" ]
      then
      missing="both the 'Editing Package' and the
      'Network Support Utilities Package (1.1)'"
   elif [ -z "$e" ]
      then
      missing="the 'Editing Package'"
   else
      missing = "the 'Network Support Utilities Package (1.1)'"
   fi

   echo "Error: Before you can install the <name of your application>"
   echo "package, you must install $missing"
   echo "from your UNIX System V/386 Foundation Set.  Installation"
   echo "terminated."
   echo
   echo "Strike ENTER to continue."

   read anything
   exit 1
fi
... # rest of your Install script
```

In the example scenario, since the user installed the Editing Package but not the Network Support Utilities Package, he/she would see the following message when your **Install** script was first executed:

```
Error: Before you can install the <name of your application>
package, you must install the 'Network Support Utilities Package (1.2)'
from your UNIX System V/386 Foundation Set.  Installation
terminated.

Strike ENTER to continue.
```

Remove Program

The interface to the removal process is provided through the **removepkg** program. This invokes the application **Remove** program, generally a shell script. The **Remove** program, once executed, should leave the system in the same state it would have been in had the new application never been installed. Depending on the situation, files created by the application may also need to be removed. This is, however, something for the user to decide, and it is recommended that users be consulted before attempting to remove any user-owned files. A sample **Remove** program can be found in the section titled "Remove Fruit Shell Script".

Examples of Scripts

The following pages show an example of an **Install** script and a **Remove** script. Each program must be custom tailored to the application it supports, but these programs should serve as guidelines.

Install Fruit Example

```
#       Install Fruit
#
LINKFILES="orange apple"
TEXTFILES="orange.txt apple.txt pear.txt"
chmod 644 $TEXTFILES
chmod 555 pear

echo "Installing Fruit Commands"
mv pear /usr/bin > /dev/null 2>&1

for i in $LINKFILES
do
        ln /usr/bin/pear /usr/bin/${i} > /dev/null 2>&1
done

if [ ! -d /usr/lib/fruit ]
then
        mkdir /usr/lib/fruit > /dev/null 2>&1
        chown bin /usr/lib/fruit
        chgrp bin /usr/lib/fruit
        chmod 661 /usr/lib/fruit
fi

echo "Installing Fruit Text"
mv $TEXTFILES /usr/lib/fruit > /dev/null 2>&1

echo "Adding Fruit to Programs Menu"
cd /usr/bin
chown bin pear
chgrp bin pear
cd /usr/lib/fruit
chown bin $TEXTFILES
chgrp bin $TEXTFILES
```

Remove Fruit Shell Script

```
#       Remove Fruit
#
USRBINFILES="orange apple pear"
FRUITTEXTS="orange.txt apple.txt pear.txt"

echo "Removing Fruit Commands"
cd /usr/bin
rm -f $USRBINFILES

echo "Removing Fruit Text"
cd /usr/lib/fruit
rm -f $FRUITTEXT
cd /

#
# 3 equals . .. and total; do not remove directory if a user put files there
if [ -d /usr/lib/fruit ]
then
        if [ 3 -eq `expr \`ls -al /usr/lib/fruit | wc -1\`` ]
        then
                echo "Removing the empty /usr/lib/fruit directory"
                rm -rf /usr/lib/fruit
        fi
fi

echo "Removing Fruit Salad from the Programs Menu"
```

Remaining Installation Files for the Fruit Package

```
$
$ cat Name
Fruit Information
$
$ cat Size
USR=40
ROOT=1
$ cat Files
/usr/bin/apple
/usr/bin/orange
/usr/bin/pear
/usr/lib/fruit/apple.txt
/usr/lib/fruit/orange.txt
/usr/lib/fruit/pear.txt
$
```

3 Device Drivers

Device Drivers

This chapter defines procedures for writing and packaging a device driver for UNIX System V/386 Release 3.2 (Version 1.0). It contains general information on "generic" UNIX System device drivers. Also described is the Installable Driver (ID) scheme for UNIX System V/386. ID allows users to add peripheral devices via a floppy diskette containing a Driver Software Package (DSP). Users will install and remove DSPs by using the **installpkg** and **removepkg** commands. This chapter also provides the implementation-dependent information for UNIX System V/386. Additional generic driver reference material can be found in the *Device Driver Reference Manual*.

It is assumed that the reader has user-level experience with the UNIX System, some general knowledge of UNIX System concepts, and the ability to write sophisticated C language programs. Writing a device driver carries a heavy responsibility. As part of the 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.

What is a UNIX Device Driver?

The 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 physical devices, such as terminals, disks, tape drives, and network media. To simplify the terminology, this chapter 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 UNIX System is the ease with which new hardware can be integrated with existing software. The integration process is simple because the operating system architecture provides a uniform software interface to every device. Processes use the same model when communicating with disks, terminals, printers or even "pseudo" devices that exist only in

software. Every device on a 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 UNIX System V/386:

```
crw-rw-rw  1 root  root    7,  1 Nov 26 12:33  lp
```

This says that the "lp" (lineprinter) 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.

The Generic UNIX Driver

This section addresses issues relevant to drivers on any UNIX System. Throughout this section, references are made to how things work on a "generic" or traditional UNIX System, along with some specific details on how UNIX System V/386 is implemented. The areas of device interrupts and priority levels in particular are heavily machine-dependent and reflect UNIX System V/386 implementation.

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.

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 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 UNIX System device drivers may find the implications of a multi-tasking environment more complex than expected.

System Buffers

A feature common to most drivers is their use of buffers. There are two types of buffers in a standard UNIX System V/386 kernel: system buffers and clists. They differ greatly in size and structure and are meant to fulfill different needs.

System buffers are the size of a file system block, on UNIX System V/386, 1024 bytes. This buffer pool primarily supports disk I/O operations. The clist manages groups of buffers of much smaller size, typically holding only 64, 128 or 256 bytes each. They were created to support I/O typified by lower data rates (for example, terminal I/O). While drivers may allocate their own data areas or independent buffer pools, this increases the size of the driver, and thus the size of the kernel.

The buffers are a commonly used UNIX System resource. The pools are of fixed sizes, though the number of buffers is controlled by constants in the kernel. Whether it uses a private buffer or the public pools, every driver should be written with the finite nature of the machine in mind; space used for buffering is taken away from user processes, so intense buffer use by a driver can reduce the performance of other drivers or require more memory be devoted to buffers. If more memory must be allocated to buffers, this decreases the memory available for user processes. More will be said later

about how to obtain and return buffers.

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 engenders. 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 the 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.

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 UNIX System V/386 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 process should not keep the machine idle while another process is ready to run, so some way must be found to have the first process wait until its information 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 a number, 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 number was used by the process as the sleeping event to wake it. This scenario for coordination between asynchronous events appears throughout the kernel.

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," one hundredth of a second. The defined constant *HZ* gives the line frequency used by a given kernel. 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. This data structure is a list of events in a simple array. Insertion of the event requires copying all elements of the list following the inserted event.

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.

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. Drivers provide the connection between two frames of reference: the process and real time realms.

The portion of the driver that deals with real time events is driven by interrupts from devices, and is thus called the interrupt section. 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 and in the context of any process. It cannot engage in any activity that depends on

process context.

Interrupt Processing

The previous section defined the interrupt and synchronous portions of a driver and mentioned that the interrupt portion is driven by real time events. The events are demands for attention from the controlled devices.

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().

Critical Sections of the Driver

The discussion so far has been centered around a particular interrupt, occurring in isolation. Though helpful, this view is unrealistic and potentially misleading. Interrupts from all devices on the system can occur at any time, and the 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 may be caught in the midst of its work. This could 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 refer 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 the section "Setting Processor Priority Level." 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. Finally, realize that the manipulations do not take place in a single machine operation.

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. The fact that the interrupting routine uses it is sufficient, proving any portion of code that can be interrupted and that also 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.

How Data Moves Between the Kernel and the Device

The discussions above assume that the data moves magically 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 or to set up a block transfer between the I/O device and memory. This process is often called 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 UNIX System V/386, 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. UNIX System V/386 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 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 the appendices:

```
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, 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.

DMA Allocation Routines

A DMA controller has control registers defining the DMA start address and word count that the driver must manipulate. See the section "DMA Controller Operations." These routines allow DMA usage to be interlocked 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.

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 suffer from 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 Appendix A) to set DMAEXCL to 0. Legal values are 0 and 1.

The names of the various channels are defined in the file **dma.h**.

dma_alloc (*channel, mode*)

Purpose: This routine allocates a DMA channel.

Parameters: *channel* is the channel to be allocated. If *mode* is DMA_NBLOCK, the routine will not sleep until the specified channel is available, but instead return a non-zero value immediately. If *mode* is DMA_BLOCK, the routine will sleep until the channel is available. This routine may only be called at interrupt time if DMA_NBLOCK is specified.

Result: Returns 0 if the channel is allocated; otherwise, returns 1.

dma_relse (*channel*)

Purpose: This routine releases a DMA channel that was either allocated with **dma_alloc()**, or implicitly allocated by **dma_start()**. It should be called as soon as the DMA transfer completes.

Parameters: *channel* is the channel number that was presented earlier to **dma_alloc()** or **dma_start()**.

Result: No return value.

dma_start (*dmareqptr*)

Purpose: This routine starts up a DMA request. It is designed to be used at interrupt time. When the channel is available, the *d_proc* routine will be called at spl6( ), with a pointer to *d_params* as an argument. The *d_proc* and *d_params* values are found in the structure pointed to by *dmareqptr*. The routine specified by *d_proc* must follow all the normal rules of UNIX System interrupt routines. The routine should be minimal because it may be called during some other device's interrupt routine.

Parameters: The *dmareqptr* structure is defined as follows:

```
struct dmareq {
        struct dmareq   *d_nxt;
        unsigned short   d_chan;
        unsigned short     d_mode;
        paddr_t           d_addr;
        long              d_cnt;
        int             (*d_proc)();
        char             *d_params;
} *dmareqptr;
```

The d_nxt field is used to link the structure onto a list of dmareq structures in case it can't be serviced immediately. The d_mode field supplies the direction of the transfer: it is either *DMA_Wrmode* (from memory to the device) or *DMA_Rdmode* (from the device to memory). The d_addr field contains the physical address from which or to which to transfer. The d_cnt field contains the number of bytes or words to transfer. The *d_proc* routine will be called at priority spl6( ) when the channel is available.

Result: Returns 1 if the request was completed immediately or 0 if it was queued for later execution.

dma_param (*channel, mode, addr, cnt*)

Purpose: This routine masks the DMA request line on the controller, sets the address and count parameters, and sets the mode (read or write).

Parameters: *channel* is the channel number that was earlier presented to **dma_alloc()**. *mode* is either *DMA_Wrmode* for a write transfer (from memory to the device), or *DMA_Rdmode* for a read transfer (from the device to memory). *addr* is the physical address from which or to which to transfer. *cnt* is the number of bytes (minus one) to transfer.

Result: The controller is initialized.

dma_enable (*channel*)

Purpose: This routine clears the mask register on the controller to allow the DMA transfer to begin.

Parameters: *channel* is the channel number that was earlier presented to **dma_alloc()**.

Result: The transfer will take place.

longdma_resid (channel)

Purpose: This routine returns the number of bytes that were not transferred by the previous **dma_enable()** request as a long.

Parameters: *channel* is the channel number that was earlier presented to **dma_alloc()**.

Result: A long integer expressing the number of bytes not transferred will be returned.

UNIX System Driver Specifics

Types of Devices

There are two classes of devices: block and character. Block devices are addressable. As the term implies, the data on the device is formatted and addressed in "blocks." The term "character device" is a misnomer that should be "raw device," implying that the data being read is raw or unformatted; the device drivers and user programs assign semantics to the data, not the file system. A device could be both a block and character device in a system configuration, implying that the system can access the device in two ways.

Although device drivers are normally associated with hardware 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.

Appendix C contains a sample trace driver as a device driver model. You may actually use this driver to help debug the driver you are developing.

Special Files

The 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 UNIX System files (such as **/etc/passwd**) are therefore the same calls that access devices (such as **/dev/console**). The system calls are open(), close(), read(), write(), and ioctl(). The section "Function Specifications (Driver Entry Points)" describes the system calls at the driver level in detail.

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 (with an upper limit of 64 for character devices and a limit of 32 block devices for UNIX System V/386). 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 (for example, 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 UNIX System accesses different subdevices using the different minor numbers.

In traditional UNIX Systems, major numbers were assigned by the driver writer or the system administrator. The **mknod** command was then used to create the files (or nodes) to be associated with the device. The UNIX System V/386 ID feature assigns the major number when the DSP is loaded by the user. More will be said about this later.

The /dev Directory

The device file may exist anywhere in the file system, but 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. 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 UNIX System V/386 and other 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.

The Master and System Files

Associated with device drivers are two device configuration files: the Master file and the System file (also known as the **dfile**). For UNIX System V/386, 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 Appendix A and *mdevice*(4) and *sdevice*(4) in the *Programmer's Reference Manual* for information describing the **mdevice** and **sdevice** file format.

The **mdevice** file contains the device name (8 characters or less), definition of what functions the device supports (field 3 has an "r" if the *read* function is implemented, has a "w" if *write* is implemented, etc.), definition of block and/or character major number, and other descriptive information about the driver.

The **sdevice** file contains information on how the device is installed in the system, that is, the number of units (subdevices), interrupt vector number (IVN) used, and other local information.

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 of the form *extern*, 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 UNIX System V/386, 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 that ID executes will set TR_UNITS equal to 4 based on the *unit* parameter (field 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. For example, changes to the process table usually require recompilation of the **ps** command. Driver programmers should likewise refrain from tampering with kernel source files.

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:

1. **/usr/include/sys/types.h** - basic system data types
2. **/usr/include/sys/param.h** - fundamental system parameters
3. **/usr/include/sys/dir.h** - directory structure definition
4. **/usr/include/sys/user.h** - the user structure definition

   The driver must include **dir.h** and **user.h** if the error field u.u_error is set in the driver or if the fields u.u_base or u.u_count are used (see the section "Read and Write"). The error field gives error

information to the kernel, and the information later returns to the user program. The introduction to Section 2 of the *Programmer's Reference Manual* describes the values the field may take. If the driver includes **user.h**, it must first include **dir.h** because of interdependencies between the two header files.

5. **/usr/include/sys/signal.h** - definition of system signals

   If the driver sends signals to user processes, it must include this file.

6. **/usr/include/sys/conf.h** - definition of device switch tables

   This file is needed if the driver uses line disciplines (see the section "Use of Line Disciplines").

7. **/usr/include/sys/file.h** - definition of file structure

   This file is needed if the driver uses control flags such as "no delay" (FNDELAY).

8. **/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").

9. **/usr/include/sys/tty.h** - definition of the clist structure

   This file is needed if the driver uses clists (see the section "Clists").

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 Appendix C, 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 "drive" 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, etc. 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.

Function Specifications (Driver Entry Points)

This section describes the functions that form the driver interface to the kernel. For a raw device, they are init, start, open, close, read, write, ioctl, poll, and halt. For a block device, they are init, start, poll, halt, open, close, and strategy. A driver need not contain every routine if one or more are irrelevant (a lineprinter 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.

Init

This routine, if present, is called to initialize the device when the operating system is first booted. It is called indirectly through its entry in the *io_init*[ ] table.

```
nnetinit ( )
```

All **init** routines are called while interrupts are still disabled (*splhi* priority). These routines should not lower the priority level.

Start

This routine, if present, is also called when the operating system is first booted. It is called indirectly through its entry in the *io-start*() table. The **start** routines are called after all **init** routines have been called and after interrupts have been enabled.

```
nnetstart( )
```

Open

The kernel calls the driver open function as a result of an open system call for the device file.

```
nnetopen(dev, flag, otyp)
int dev_t dev, flag, otyp;
```

The parameters of the driver open function are the minor device number of the device file, and the flags supplied in the "oflag" field of the open system call (see Section 2 of the *Programmer's Reference Manual*) correspond to flag values in the header file **file.h**. The minor device number usually corresponds to the unit number of the physical device being opened. The responsibility of the open routine is to establish a "connection" between the user process issuing the open call and the device being opened. It is impossible to be more specific since devices have different ideas of what "being opened" or "being connected" is. The driver must follow the specifications of the manual for the open system call regarding the O_NDELAY flag: if set, the open must return without waiting for a hardware connection and if clear, the open must block (see the section "Sleep and Wakeup" on sleep()) until the hardware establishes a connection. Error conditions should be set in the field u.u_error. The usual error condition is ENXIO, indicating that the device does not exist (out of range). If it does not make sense for two processes to open one device unit, the second open should return with an EBUSY error.

The "otyp" field defines open/close protocol rules to help the driver know when the "last close" occurs. See "open.h" for the flag values and additional information.

Close

The kernel calls the driver close function with the minor number of the device file as its parameter.

```
nnetclose(dev, flag, otyp)
int  flag, otyp;
dev_t dev;
```

The responsibility of the close function is to end the connection between the user process and the previously opened device, and to "clean up" the device (hardware and software) so that it is ready to be opened again.

Note that the kernel calls the driver open function on every open system call executed by user programs. However, the close function is usually called only

on the last close system call (see "otyp" rules in "openh"); if, for example, a character device is opened by more than one user process, and they then all close the device and exit, the driver will see several opens, but only one close.

Read and Write

The kernel calls the driver read and write routines to read (write) data from (to) the device specified by the unit number, the only parameter.

```
nnetread(dev)
dev_t dev;
```

and

```
nnetwrite(dev)
dev_t dev;
```

Drivers for "raw" devices contain these routines. Because user programs and the operating system execute in different address spaces, the I/O cannot take place directly from the device to the user program (unless the device is also a "block" device, as will be explained at the end of this section). There must be a system buffer between them. When reading, the driver must receive the data from the device in a read buffer and then copy the data from the buffer to the user process's local buffer. When writing, the driver must copy the data from the user process's local buffer and then transmit the data from the system buffer to the device. The system buffer can be a private driver data structure or one obtained by use of the system utility routines described in the section "Allocating Buffer Space."

In the driver read routine, the system variable u.u_base is the address of the buffer in the user program address space, and the variable u.u_count is the number of bytes remaining to be read. The functions copyout(), subyte(), or suword() should be used to copy the data from the driver buffer to the user buffer:

```
copyout(ptr_to_driver_buffer, u.u_base, n)
subyte(u.u_base, char_c)
suword(u.u_base, long_w)
```

where n is the number of bytes the function copies from the buffer (pointer) ptr_to_driver_buffer in the copyout() function, char_c is the single character that subyte() copies to the user buffer, and long_w is four bytes copied by suword(). The driver should use copyout() for copying more than a few bytes of data. After the function calls, the driver should increment the value of u.u_base by n and decrement the value of u.u_count by n, the number of bytes transferred. For subyte(), the value of n is 1. For suword(), n is 4. If

either function returns a non-zero value, then u.u_error should be set to EFAULT to indicate the error.

The driver write routine is similar to the read routine, except that the routines copyin(), fuword() and fubyte() are used to copy data from the user buffer to a system buffer:

```
copyin(u.u_base, ptr_to_driver_buffer, n);
fubyte(u.u_base);
fuword(u.u_base);
```

The system is not responsible for "bad" addresses set in u.u_base (set as a result of the user system call). If the user is reading in 512 bytes from the device into a user data structure that is 256 bytes long, it is not the system's job to detect the error. The user program must make sure that the data returning from a read system call will not overflow the user buffer.

The copyio() routine can be used in place of the copyin() and copyout() routines, if desired. The copyio() routine copies bytes to and from a physical address (that is, buffer address) in the kernel to and from a long address (that is, user data pointer). The syntax for copyio() is as follows:

```
copyio(addr, caddr, cnt, mapping)
paddr_t addr;
caddr_t faddr;
unsigned cnt;
int mapping;
```

The argument *addr* is a pointer to the physical kernel address to which or from which the data is to be transferred.

The argument *caddr* is a 32-bit pointer that contains the offset of the user address to which or from which the data is to be transferred.

The argument *cnt* is an unsigned integer that specifies the number of bytes of data to transfer.

The value of *mapping* is an integer that designates the direction of the transfer. The following possible mapping values are defined in **user.h**:

U_WUD	transfers from user data to kernel data (buffer)
U_RUD	transfers from kernel data (buffer) to user data
U_WKD	transfers from kernel data to buffer
U_RKD	transfers from buffer to kernel data

If successful, this routine performs the specified data transfer; otherwise, it

returns -1.

The read and write routines are responsible for resetting the device hardware so that later calls work correctly. They are also responsible to "clean up" errors and keep appropriate statistics.

Raw devices that are also block devices can avoid copying data into an intermediate buffer by using the physio() routine and the strategy() routine:

```
physio(strategy, bp, dev, rdwr_flag);
int (*strategy)();
struct buf *bp;
int dev;
int rdwr_flag;
```

The strategy parameter is the name of the strategy() routine for the device, described in the next section. The buffer header pointer, bp, is a locally allocated buffer header, not one allocated from the buffer pool (as in the section "Buffer Pool") because the physio() routine assigns the data pointer in the buffer to the location in the user program (u.u_base) where the data transfer should come from or go to. If the bp parameter is null, the physio() routine assigns a buffer internally (probably a safer way to invoke it). The dev parameter is the device number, and the flag rdwr_flag should be B_READ or B_WRITE, as appropriate. The physio() routine will call the device strategy routine internally and set up the data transfer directly between the device and user space. This can only be done if the device is a block device as well as a raw device. A prototype Floppy Disk Driver in Appendix D contains an example of physio() usage.

Strategy

Drivers of block devices must define a strategy routine to start I/O to and from the devices:

```
nnetstrategy(bp)
struct buf *bp;
```

The kernel calls the strategy routine with a parameter that is a pointer to a buffer header containing all the information about the I/O operation. For example, the definition of a strategy function for the disk driver called dsk would look like

```
#include "sys/buf.h"

dskstrategy(bp)
struct buf *bp;
{
 /* body of strategy routine */
}
```

The strategy routine uses the following fields in the buffer, but it should not set them:

1. dev_t b_dev;

   b_dev contains the major and minor number of the device where the I/O is to occur. The minor number is contained in the 8 low order bits, and the major number is contained in the next 5 low order bits. The 3 high order bits should not be used.

2. daddr_t b_blkno;

   b_blkno is the block number of the device where the I/O is to occur.

3. unsigned int b_bcount;

   b_bcount is the number of bytes to be transferred by the I/O operation.

4. caddr_t b_un.b_addr

   b_un.b_addr is the address of the data in the buffer. The data array is SBUFSIZE bytes long.

5. int b_flags

   b_flags gives the buffer status. If the B_READ bit is set, then the I/O operation is to read from the device; if the B_WRITE bit is set, then the I/O operation is to write the device.

If the strategy routine finds an error in setting up the I/O or if the device reports an error via an interrupt, the driver should set the following fields:

1. b_flags should have the B_ERROR bit or'ed in. The driver should not assign a value to b_flags because that may erase other bit patterns that the kernel relies on.

2. char b_error

   b_error should be set to an appropriate error value. Typical values are EIO for some physical I/O error, ENXIO for attempting I/O on a bad device or bad device address, or EACCES for attempting to access a device illegally. The kernel later sets u.u_error with the value of b_error so that any appropriate value for u.u_error could be set.

3. unsigned int b_resid;

   b_resid should be set to the number of bytes that have not been transmitted.

Ioctl

The driver ioctl routine controls hardware parameters and the interpretation of data as it passes through the driver via read and write:

```
nnetioctl(dev, cmd, arg, mode)
int cmd, arg, mode;
dev_t dev;
```

The routine takes four parameters, all integers:

1. dev - The minor device (unit) number.

2. cmd - A command argument that the driver ioctl function interprets as the type of operation the driver should perform. The command types vary across the range of devices, but the user manual specifies the command types that must work for terminals (see the *User's/System Administrator's Reference Manual* Section 7, *termio*(7)). Drivers that use an ioctl function typically have a command to "read" the current ioctl settings and at least one other that sets new settings. The kernel does not interpret the command type, so a driver is free to define its own commands.

By convention, ioctl commands are set to complex numbers to help guard against accidental misuse by users. A common technique is to pick the first letter of the device name and left shift the ASCII code by 8, then "or" in a command code into the lower 8 bits. The values for ioctl commands are usually defined in the driver header file so that both the driver and user programs can access the commands via #defines. See **trace.h** in Appendix C for an example of ioctl assignments.

3. arg - An arbitrary argument that can pass parameters between a user program and the driver. The argument can be the address of a structure in the user program that contains settings for the driver or hardware. The driver reads the settings from the user program via the copyin() function and does the appropriate operations. Similarly, the driver collects current settings and uses the copyout() function to write the data into the user program structure. Alternatively, the argument may be an arbitrary integer that has some meaning to the driver. Except for terminal drivers where the *User's/System Administrator's Reference Manual* specifies the argument values (see Section 7, *termio*(7)), the interpretation of the argument is driver-dependent and usually depends on the command type; the kernel does not interpret the argument.

4. mode - An argument (need not be used) that contains values set when the device was opened. The driver can use the mode to determine if the device unit was opened for reading or writing, if necessary, by checking the FREAD or FWRITE setting.

Code for Bringing a Device into Service

There may be requirements for writing initialization code to bring a device into service. Often this can be done in the open routine by creating a flag to determine if this is the first open (generally after a reboot of the system) of the device. If such code is necessary, the driver should contain an initialization function that follows the naming convention described earlier. For our example, nnet, the initialization function would be called nnetinit().

If the initialization routine must be called at system initialization time, the Master file entry for the driver should contain an "I" in field 3 in addition to the other driver functions that are supported.

Poll

The routine nnetpoll, if present, is called by the system clock at spl6() during every clock tick. It is useful for repriming devices that constantly lose interrupts.

```
nnetpoll(ps)
int ps;
```

The parameter ps is an integer that indicates the previous process's priority when it was interrupted by the system clock.

Halt

The routine nnethalt, if present, is called when the system is shut down.

```
nnethalt()
```

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.

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.

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 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 UNIX System V/386, there are a limited number of available interrupts. For more information on this and other machine-dependent aspects of the UNIX System V/386 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(ivn)
int ivn;
```

where ivn indicates the interrupt number associated with the interrupt. 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 the 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().

Sharing Interrupts and DMA Channels

The UNIX System V/386 Installable Driver (ID) scheme allows for the sharing of interrupt lines and DMA channels among device drivers. When an interrupt occurs, the interrupt handler for each device sharing the interrupt is 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, field 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 Appendix A and the *Programmer's Reference Manual* for manual pages describing the sdevice file format.

To indicate that a device can share its DMA channel, field 3 of the mdevice (the characteristic field) entry must include a **D** identifier. All drivers sharing DMA channels must include a **D** in field 3 of their mdevice entry. If they do not, an error will result during kernel configuration. See Appendix A and the *Programmer's Reference Manual* for manual pages describing the mdevice file format.

Use of Line Disciplines

Line disciplines are modules that interact with a driver to control the data as it passes between the kernel and the device driver. The driver controls the hardware, but the line discipline module controls software manipulation of the data. It would be natural to think of networking drivers such that the networking driver controls the hardware medium, and the network protocol is contained in a line discipline. But historically, line disciplines have been associated mostly with terminals. Although used by the UNIX System V/386 console and "asy" drivers, detailed discussion of line disciplines is beyond the scope of this chapter. There is, however, additional information on the ttin(), ttinit(), ttioctl(), ttopen(), ttread(), ttwrite(), etc. line discipline kernel functions in the *Device Driver Reference Manual*.

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.

System Utility Functions

The driver calls kernel routines to perform system-level functions, many of which were introduced in the section "Driver Activities and Responsibilities." The following paragraphs describe the syntax and use of these kernel functions.

Sleep and Wakeup

As described in the 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 does a context switch and schedules another process.

The sleep function takes two parameters: the address (signifying an event) upon which the process will sleep and a priority value that is assigned to the process when it is awakened:

```
sleep(addr, pri)
caddr_t addr;
int pri;
```

The address used for sleeping is an arbitrary address 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(addr)
caddr_t addr;
```

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. 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 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().

Setting Processor Priority Levels

As described in the 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 the processor execution level 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 UNIX Systems have an splhi() function to set the processor execution level to the highest value, which is spl7() for UNIX System V/386.

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 in that case but rather to the old priority level. Consider the following code:

```
register int s;
.
.
s = spl5();
while ((cp = getcb(&tp->t_rawq)) != NULL)
      putcf(cp);
tp->t_delct = 0;
splx(s);
```

Particularly 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:

```
spl5();
driver.state |= condition;
while (driver.state & condition)
        sleep(&driver.state, PRIORITY);
spl0();
```

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, UNIX System V/386 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 spl7()). A device at IPL6 can be interrupted by a device at IPL7 or IPL8. In UNIX System V/386, 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 lineprinter is more safely interrupted and is given a low IPL. See the section "Controller Interface Basics" for a more complete definition of the device configuration assignments.

```
DEV     IPL     Device attached
---     ---     ------------------------------
clock   8       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
```

Care must be taken not to overstate device interrupt priority and to limit the amount of time spent at high levels. For example, if any driver elevates to spl7() for more than a few milliseconds, loss of UNIX System clock time may result.

Sleep Priorities

The second parameter to the sleep() function, a scheduling parameter used when the process awakens from its sleep, must be a constant and not a variable. The parameter, called the sleep priority, has critical effects on the sleeping process's reaction to signals. If it is lower than the manifest constant PZERO (25 on most systems), that is, it is higher than PZERO (lower value priority levels mean higher priority in the UNIX System), 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 priority greater than PZERO (numerically, 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 less than PZERO (numerically 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.

Priority values range between 0 (highest priority) and the constant PUSER (lowest system priority, usually around 60). 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" before doing the longjmp() on receipt of a signal while sleeping. 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 doing the longjmp():

```
if (sleep(sleep_address, condition | PCATCH))
{
        /* driver code cleanup */
        .
        .
        .
        longjmp(u.u_qsav, 1);
}
```

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);
        longjmp(u.u_qsav, 1);
}
/* somebody woke up driver...
 * continue normally here */
```

The kernel saves the field u.u_qsav for use in a longjmp() function call. No other parameter should be used, nor should the driver contain a setjmp() function call. The second parameter of the longjmp() function call should always be 1.

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

```
timeout(repeat, n, count);
```

where n is the parameter to the function repeat(), to be called after 'clock' 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 where the function repeat() is defined.

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. Specific information on the malloc(), mfree(), and mapinit() kernel functions can be found in the *Device Driver Reference Manual*.

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 UNIX System provide buffer space.

Buffer Pool

The 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 size of UNIX System V/386 buffers is 1024 bytes. The functions that drivers may use to manipulate the buffers are

1. struct buf *geteblk();

   Allocates a buffer and returns a pointer to a buffer header that, in turn, points to the data buffer.

2. brelse(bp) struct buf *bp;

   Releases a previously allocated buffer.

3. iowait(bp) struct buf *bp;

   Sleeps on the buffer awaiting an event, such as completion of I/O.

4. iodone(bp) struct buf *bp;

   Awakens a process sleeping via iowait().

5. 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 are

1. B_WRITE when writing data from the buffer to the device.

2. B_READ when reading data from the device.
3. B_DONE, set by the function iodone(), to indicate that the I/O operation has completed.
4. B_ERROR to indicate an error in use of the buffer.
5. 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;
```

6. 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) {
                iowait(bp);
                if(bp->b_flags&B_ERROR)
                        rcode = -1;
                bp->b_flags &= ~B_CNTL;
                brelse(bp);
        }
        return(rcode);
}
```

Clists

By including the header file **sys/tty.h**, drivers can use clists and cblocks (generic names for character lists and character blocks) to buffer small bursts of data from slow-speed devices. Drivers should use clists if they are interested in character by character processing of data, as in terminal drivers. The only field in the cblock that a driver may access directly is

```
char    c_delim;
```

The driver can use it to record status of the cblock. The size of the cblock data buffer is CI SIZE bytes, usually set between 64 and 256 bytes, as compared to the buffer sizes of 1024 bytes.

The driver should not access fields in the clist or cblock data structure (except for c_delim) unless it uses the following routines:

1. getc(p) struct clist *p;

   Returns a character (really an int) from the clist pointed to by p, but it returns -1 if the clist is empty.

2. putc(c, p) char c;

   struct clist *p;

   Places the character at the end of the clist pointed to by p. If system resources are exhausted, putc returns -1 (error); otherwise, it returns 0.

3. struct cblock *getcf()

   Returns a new cblock to the caller, returning NULL if no cblocks are available in the system.

4. putcf(cp) struct cblock *cp;

   Returns the cblock pointed to by cp to the system.

5. struct cblock *getcb(p) struct clist *p;

   Returns a pointer to the first cblock on the clist p, but it returns NULL if the clist is empty.

6. putcb(cp, p) struct cblock *cp; struct clist *p.

   Places the cblock pointed to by cp on the end of the clist p.

7. getcbp(p, cp, n)

   Copies characters from the specified clist, p, to the buffer addressed by the cp argument. cp is a char * addressing the buffer to which the characters are to be copied. n is the number of characters to be copied. getcbp must be called at spl6(). This routine returns the number of characters actually copied, which is less than or equal to n.

8. putcbp(p, cp, n)

   Copies characters from a buffer to the clist given as an argument. p is a struct clist *. cp is a char *, which addresses the buffer. n is the

number of characters to be copied to the clist. putcbp must be called at spl6().

Here is an example of the use of clists, taken from a routine to read the "canonical" input from a terminal (note that the routine is not given here in its entirety). The tty structure contains the clists t_canq and t_rawq:

```
canon(tp)
register struct tty *tp;
{
        register struct cblock *cp;

        spl5();
        if (tp->t_rawq.c_cf == NULL)
                tp->t_delct = 0;
        while (tp->t_delct == 0) {
                if(!(tp->t_state&CARR_ON) || (u.u_fmode&FNDELAY)) {
                        spl0();
                        return;
                }
                tp->t_state |= IASLP;
                sleep((caddr_t)&tp->t_rawq, TTIPRI);
        }
        if(!(tp->t_lflag&ICANON)) {
                tp->t_canq = tp->t_rawq;
                tp->t_rawq = ttnulq;
                tp->t_delct = 0;
                spl0();
                return;
        }
        spl0();
        while((cp = getcb(&tp->t_rawq)) != NULL) {
                putcb(cp, &tp->t_canq);
                if(cp->c_delim)
                        break;
        }
        tp->t_delct--;
}
```

UNIX System V/386 Installable Driver Implementation

This section describes the UNIX System V/386 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 **installpkg** command delivered in the Base System Package of the UNIX System V/386 Foundation Set. Driver developers must use the C Programming Language Utilities (CPLU) delivered in the C Software Development Set of the UNIX System V/386 Software Development Set to compile their driver and build installation scripts for delivery with the device driver (floppy diskette) package. The section "Device Driver Development Methodology" provides step-by-step procedures on how to write, compile, debug, and finally package the device driver.

Users will be exercising ID when adding new peripheral boards 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 UNIX System at user level, then has the user reboot the system using the new kernel as is currently done on many other 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, an install script, a remove script, and device-specific entries for system configuration, initialization and shutdown files, as well as space allocation entries normally associated with **space.h** on earlier 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 UNIX System V/386 Base System Set software.

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)
- a port address range through which the CPU and device can communicate (IOA - I/O address)
- 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)).

Interrupts

In addition to the 80386 processor chip, most 386 computer systems are outfitted with two Intel 8259 peripheral interrupt controllers (PIC), each with 8 interrupt lines. The 16 interrupt ports of the controller are assigned as follows:

interrupt number	buss pin	common name	devices on 386 systems using interrupt
0	-	clock	1/100 second timer
1	-	keyboard	keyboard
2	-	game port	expansion PIC (see IVN 9)
3	IRQ3	com2	serial port 2
4	IRQ4	com1	serial port 1
5	IRQ5	hard disk	not used
6	IRQ6	floppy	integral floppy controller
7	IRQ7	printer	integral parallel port
8	IRQ8	-	real time clock
9	IRQ2	-	not used (wired to IVN 2)
10	IRQ10	-	not used
11	IRQ11	-	not used
12	IRQ12	-	not used
13	IRQ13	-	iAPX387 math co-processor
14	IRQ14	-	integral hard disk controller
15	IRQ15	-	not used

In the above table, IVN 0 through 7 have a "common name" that is derived from the PC/XT architecture. Most 386 systems have a PC/AT architecture with 15 available interrupts. IVN 2 is used to connect the second PIC; however, peripheral boards that use IRQ 2 can still be used by configuring the device driver to expect IVN 9. Note that with the expansion PIC installed, the hard disk is moved to IRQ14, freeing up IRQ 5 for PC/XT add-on devices.

Most devices that require an interrupt are hardware strappable to 2 or more different interrupts to allow the user some flexibility in installation.

I/O Addresses and Controller Memory Addresses

Each controller requires an IOA and possibly a CMA. These address regions must be unique and not overlap with any other device's address regions. Your Hardware Technical Reference Manual should show that IOA and CMA addresses are permanently assigned to the above list of devices and to some optional peripheral devices. If a device on the parent board is not configured into a kernel, the interrupt is freed up, but the IOA and CMA

remain assigned to that device and should not be used by any new device.

A quick look at the file **/etc/conf/cf.d/sdevice** shows assignments for the base system. The IVN, starting and ending IOA and CMA addresses for the UNIX System V/386 without any added peripheral boards, is as follows (IOA and CMA values are in hexadecimal):

Device	Prefix	IVN	SIOA	EIOA	SCMA	ECMA
Serial ports	asy	4	3f8	3ff	0	0
	asy	3	2f8	2ff	0	0
Floppy disk	fd	6	3f0	3f7	0	0
Co-processor:	fp	13	0	0	0	0
Hard Disk:	hd	14	320	32f	0	0
keyboard:	kd	1	60	64	0	0
Parallel Port:	lp	7	378	37f	0	0
Real Time Clk.:	rtc	8	0	0	0	0

DMA Controller Operations

Most 386 computer systems have two Intel 8237A DMA Controllers, which provide seven channels to transfer data directly to and from memory without CPU involvement. The DMA Controller hardware should be described in your Hardware Technical Reference Manual. The DMA controllers are accessed through a collection of control registers mapped to I/O (Port) addresses. The following table is a summary of DMA channels and their usage in the base system. Examine the file **/usr/include/sys/dma.h** for additional information on control register locations used to initiate DMA.

Ch 0: spare
Ch 1: spare
Ch 2: floppy
Ch 3: spare
Ch 4: unusable - cascade from chip 1
Ch 5: spare
Ch 6: spare
Ch 7: spare

Channels 0-3 support 8-bit transfers, while channels 5-7 support 16-bit transfers. These 16-bit channels can transfer up to 128K at a time since counts are in 16-bit words instead of 8-bit bytes. The page registers for channels 5-7 control address bits A17-A23 (instead of A16-A23); therefore, DMA boundaries exist at multiples of 128K instead of 64K. These 16-bit channels cannot transfer data on odd byte boundaries.

User Interface

The ID allows a user to install or remove device drivers using **installpkg** and **removepkg.** The **installpkg** command installs a DSP from a floppy diskette onto UNIX System V/386 and initiates automatic procedures to reconfigure the kernel. The **removepkg** command allows the user to select which package to delete. It then removes the DSP from UNIX System V/386 and reconfigures the kernel without the driver.

The **displaypkg** command displays any software packages that the user has installed. DSPs are treated identically to other UNIX System V/386 Software packages. Device drivers that are pre-installed on the system by the Base System Set floppy diskettes are not displayed by this command.

Each add-on DSP must contain an Install script that will

- optionally prompt the user for any necessary IVN, IOA or CMA information required due to hardware re-strapping
- check for any conflicts with other drivers (for example, IVN, IOA, or CMA conflicts)
- reconfigure the kernel with the new driver

User Privileges

The ID uses the same installation rules as any other UNIX System V/386 add-on software, that is, the user needs *root* permissions. A user must be *super-user* to install DSPs.

Interactions with Other UNIX System V/386 Processes

The ID 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.

Number of Installed Drivers

Due to limited available interrupts, as defined in the section "Interrupts," there is a limit to the number of conventional peripheral devices which can be installed on a UNIX System V/386. Additional drivers could, however, be installed for devices not requiring interrupts, for software pseudo-devices, or for devices sharing interrupts. (See the section "Sharing Interrupts and DMA Channels.")

As the table in the section "Interrupts" shows, there are several AT-type interrupts available but few XT-type. In that list, IVN3 is assigned to the add-on serial port (COM2), and IVN 7 is assigned to the integral parallel port (lineprinter interface). If you are installing hardware/driver software on a UNIX System V/386 that does not have a COM2 interface configured or does not use a lineprinter, it is possible to unconfigure one of those devices, thus freeing the respective IVN.

UNIX System V/386 Modifications for ID

The general architecture of the kernel components relating to device drivers and the contents of some system files was modified slightly for ID to allow UNIX System V/386 users to easily add device drivers to their systems.

Master File

In earlier 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 UNIX System V/386, 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 the manual pages in Appendix A of this document and in the *Programmer's Reference Manual.*

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 the manual pages in Appendix A of this document and in the *Programmer's Reference Manual*.

space.c

The file **/usr/include/sys/space.h** traditionally contains data structure declarations required by the kernel and device drivers. The amount of storage allocated for each driver data structure is dependent on the number of subdevices configured for a particular device. For UNIX System V/386, since there was a need to modularize storage allocation and since space allocation should rightly be done in a file, the file **/usr/include/sys/space.h** has become a collection of **space.c** files stored in the **/etc/conf** directory.

These **space.c** files determine how much storage is required for the main body of the kernel and each of the added drivers.

These files are compiled and linked into the kernel during reconfiguration.

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** - Stores all ID commands.
- **cf.d** - Contains configuration-dependent files.
  - **stune, sassign, sdevice, mdevice, mtune** - Equivalent to the master and system files of earlier UNIX Systems. The **mdevice** file is built from the *Master* modules of the installed DSPs.

- ☐ **mfsys, sfsys** - File system type information (see the *mfsys*(4) and *sfsys*(4) pages in the *Programmer's Reference Manual*).
- ☐ **vuifile** - Defines memory management definitions for the kernel.
- ☐ **init.base** - The base system part of **/etc/inittab**.
- ☐ 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 UNIX Operating System kernel; eventually to be linked to /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 two 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 **Stub.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**.

Device #defines Generated by the Configuration Process

The configuration process produces a file **config.h** that contains device parameters in the form of *#defines* 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:

```
Per device defines:
-------------------------------
#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.

Per controller defines (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.
#define  PRFX_0_DEVNM      Device number (same as PRFX_0_VECT for
                           interrupt driven devices).
```

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.

Commands for Installing Drivers and Rebuilding the UNIX Operating System Kernel

The DSP **Install** script must use calls to **idcheck**, **idinstall**, and **idbuild**. Manual pages for these commands are provided in Appendix A of this

document and in the *User's/System Administrator's Reference Manual*. Sample **Install** and **Remove** scripts, which use these commands, are provided in Appendix E.

Idcheck

This command is used to obtain selected information about the system configuration. The command is designed to help driver writers 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.

Idinstall

The **idinstall** command is called by the DSP's **Install** and **Remove** scripts, and its function is to install, remove, or update a DSP.

Idbuild Command

This command is a shell script that comprises the reconfiguration processes. **Idbuild**

- Informs the user of the approximate length of time to build the kernel.
- 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 linked to **/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 Driver Software Package

This section defines the contents of the Driver Software Package (DSP), and Appendix E contains an example package. 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 the section "Driver Development Procedures."

The DSP is to be delivered on an installation floppy diskette, as described in Chapter 2 of this *Guide*. 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.

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 UNIX System V/386 Software Development Set. The section "Device Driver Development Methodology" provides procedures for coding, compiling, and debugging the driver object module.

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 Appendix A of this document and in the *Programmer's Reference Manual*.

Fields 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.

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 Appendix A of this document and in the *Programmer's Reference Manual.*

Space.c (optional)

The file **/usr/include/sys/space.h** traditionally contains data structure declarations required by the kernel and device drivers. The amount of storage allocated for each driver data structure is dependent on the number of sub-devices configured for a particular device. For UNIX System V/386, each driver can have its own **Space.c** file containing configuration dependent-data structures.

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

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 fields can be separated by spaces. The syntax of this line is as follows:

```
Field 1: DSP internal name
Field 2: name of node to be inserted
Field 3: 'b' or 'c' (block or character device)
Field 4: minor device number

example
---------------
DSP-internal-name  node0  c  0
DSP-internal-name  node1  c  1
```

See the *idmknod*(1M) manual page in Appendix A of this document and in the *User's/System Administrator's Reference Manual.*

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 the *Programmer's Reference Manual*):

```
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 Appendix A of this document and in the *User's/System Administrator's Reference Manual.*

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 the *User's/System Administrator's Reference Manual*). When creating this module the file permissions must allow execution by **root**.

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. When creating this module the file permissions must allow execution by **root.**

Name (required)

This module contains the "external DSP name," which is the name displayed by the **displaypkg** command. The package name should be fairly specific to reduce the chance of conflicting with other packages.

If the DSP contains just a device driver, then the name **Device_Name Driver** might be a good choice. If, however, the driver is part of a package that includes a driver, communication protocols, and user commands to access a network, for example, then perhaps the word "driver" shouldn't even be used. Something like **Net_name Communications Package** may be more appropriate.

The internal name should be derived from the external name for naming **/dev** entries and for use by the **Install** script.

The internal name must be the same as the name appearing in field one of the **Master** module. The internal name must uniquely identify the driver and be eight characters or less. As an example, the Phone Manager Device Driver might have an internal name of *phone*. See the section "Driver Development Procedures" for elaboration of the internal name.

Files (required)

The DSP may contain commands, programs, or data files in addition to the ones listed as ID modules directly. It is the responsibility of the **Install** module to install or make use of these files. The **Files** module must contain a full path name of where these additional files are to be installed.

Install (required)

This module

- Installs all of the files that are listed in the module **Files**. It does not install any of the ID modules.
- Invokes the ID command **idinstall** with the **-a** option and passes it one argument, the internal DSP name. This will move the contents of the DSP to the proper directories.
- Invokes the ID command **idbuild**.

Remove (required)

This module

- Removes any commands, programs, or data files installed by the **install** 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**.

Size (required)

The contents of the **Size** file should be the number of blocks required to install the DSP. A block is defined as 512 bytes. This file is to be used by the **installpkg** program to determine if enough free disk space is available to add the driver package. At installation time, if insufficient disk space exists to install the package, the user will be alerted and the installation aborted. In addition to the space required to store the user commands and files the package is to install, a large amount of temporary space is required to reconfigure the kernel. The space required in the root ("/") file system can be roughly calculated as the size of the current UNIX Operating System kernel image (**/unix**) plus 400 blocks. If the system has a **/usr** file system, approximately 400 blocks should be specified. An example of a **Size** file is as follows (assuming a typical /unix is 1000 blocks):

```
ROOT=1400
USR=400.
```

Mfsys (optional) and Sfsys (optional)

These files are new to UNIX System V/386 Release 3.2 and specify the addition of file system types to the UNIX Operating System kernel. Additional information can be obtained from the *mfsys*(4) and *sfsys*(4) manual pages in Appendix A of this document and in the *Programmer's Reference Manual*.

Summary of Modules

Table 1 lists the modules that may appear in a DSP. The only restriction on ordering the files is that the **Size** file must be on the first floppy diskette of the DSP. See the section "Special Installation Files" in Chapter 2 for more information on installation packaging.

Module	Mandatory/Optional	Definition
Size	M	Disk space requirements to install DSP
Name	M	DSP name
Install	M	Installation script
Remove	M	Remove script
Files	M	Target paths for commands/data files
Driver.o	M	Driver object file
Master	M	Master file entry
System	M	System file entry
Space.c	O	Driver space allocation file
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 1. Components of Driver Software Package

Base System Drivers

An examination of **/etc/conf/cf.d/mdevice** will show the installed DSPs on UNIX System V/386. A partial list of the base system device drivers and software modules is as follows:

Hardware Device Drivers	
asy	Serial Ports Driver
fd	Floppy Disk
hd	Hard Disk
kd	Keyboard
lp	Line printer (Parallel Port)
rtc	Real Time Clock

Software Modules	
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

The above list does not include several drivers and software modules being packaged as add-ons such as Streams and RFS (Remote File Sharing). Drivers in the base system are installable drivers that have been delivered in the Base System Package of the UNIX System V/386 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 **displaypkg** and **removepkg** 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).

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 **Install** module. This module follows the same rules as for other driver packages except that it calls **idinstall** with the **-u** option.
- The **Name** module. The external name should include the word "update." For example, to replace the floppy disk driver with a new release, **Floppy Disk Driver Update** would be an appropriate name.
- The **Remove** 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 **displaypkg** command, the updated driver will be listed as **Device_Name Driver Update Package**. The **removepkg** menu will display the same entry, but if the user tries to select the updated driver, the **Remove** 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 **removepkg** and **displaypkg** 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.

Installation/Removal Summary

The ID commands and the DSP's modules defined above will be used together to rebuild and execute a new UNIX Operating System kernel. The step-by-step procedure to install, reconfigure, and execute a new kernel is as follows:

Installing a DSP

1. The User executes **installpkg**.
2. **installpkg** reads the floppy diskette contents and executes **Install**.
3. **Install** 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.
4. Upon return to **installpkg**, the script will call **/etc/conf/bin/idreboot**.
5. After the user confirms idreboot and presses RESET:

   The init program is the first user-level program executed after reboot; **/etc/inittab** 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 **removepkg**, and in step 2, the **Remove** script will be deleting commands and files, and the **idinstall** command will be called with the **-d** option to delete the DSP. See the *Operations/System Administration Guide* for a detailed description of this process.

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.

Appendix A of this document and the *Programmer's Reference Manual* 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.

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 UNIX System V/386 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 the *User's/System Administrator's Reference Manual* for the manual page that describes *idtune*(1M). This command will take individual system parameters, 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.

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 the *Programmer's Reference Manual*. 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.

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.

Device Driver Development Methodology

We have covered many of the kernel architectural and driver design details you need to know to write a 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 UNIX System V/386 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.

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 execute as the super-user to develop and debug device drivers. Throughout this section, the trace driver provided in Appendix C 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 UNIX System V/386 ID implementation, the following name definitions based on the internal name are required:

   - ☐ Field 1 of the Master file. This must be **DEV_NAME**.
   - ☐ Field 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 4 characters. It is desirable to make this identical to field 1 if **DEV_NAME** is 4 characters or less. For the trace driver this prefix is "tr."
   - ☐ Field 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
```

3. 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 fields 5 and 6. 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.

4. 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    trace0    c    0
```

5. 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.
6. Create a directory called **/etc/conf/pack.d/DEV_NAME**. Put **Driver.o** and **Space.c** there (if you need them).
7. At this point, it would be a good idea to make a copy of your current UNIX Operating System kernel. Execute the following:

```
cp /unix /unix.bak
```

8. 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 a "time to rebuild UNIX System" message followed either by a "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 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, and the new kernel in **/etc/conf/cf.d** will automatically be linked to **/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.
9. When the system comes up, test your driver.

Emergency Recovery (New Kernel Will Not Boot)

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 UNIX System ...` message, 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. When the prompt `Please press <RETURN> when ready to install the UNIX System` appears, press <DEL> to exit the installation program.

   You are now executing a floppy-bootable 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:

```
fsck -y /dev/dsk/0s1          # check the hard disk
mount /dev/dsk/0s1 /mnt       # mount the hard disk
cp /unix /mnt/unix            # copy a hard disk Kernel
umount /dev/dsk/0s1           # unmount the hard disk
```

   Remove the Floppy Diskette.

   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 **displaypkg** 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.

Driver Debugging

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. Printf() calls made inside the Kernel will appear on the UNIX System V/386 monitor (**/dev/console**). Note that this printf() is not the same printf in section 3 of the *Programmer's Reference Manual*. 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). The use of printf() is mentioned for historical purposes. A new general kernel error utility called cmn_err() should be used in newly written drivers. See the *Device Driver Reference Manual*.

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. The game port driver in Appendix B shows how to do this.

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 later retrieve this data:

1. 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 the *User's/System Administrator's Reference Manual* for more information.

2. 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**.

The 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.");
```

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. Appendix C 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 (see the earlier section "Clists") to save these traces. Although this driver isn't delivered with the base UNIX System V/386, you can compile and link edit the driver into your system from the source code presented in Appendix C. 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. The game port driver in Appendix B has some calls to trsave() so you can see how it is used.

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 panic() function. 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 panic() function. The argument to

the panic() function is a string that will appear on the UNIX System V/386 monitor:

```
panic("Your system has panic'ed, DEV_NAME error!");
```

The system will dump an image of memory for later analysis. If the error is recoverable, the driver should not call panic(). As with printf(), new drivers should use cmn_err() for all panics as described in the *Device Driver Reference Manual*. In the example above the correct syntax would be as follows:

```
cmn_err(CE_PANIC,"Your system has panic'ed, DEV_NAME error!");
```

Taking a System Dump

In the event a panic() occurs, there may be some value in examining the dump produced by the system. Since UNIX System V/386 uses the same physical hard disk partition 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 the *User's/System Administrator's Reference Manual* for information on how to use **crash** to examine the 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 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 UNIX Operating System
> kernel it 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.

Building the Driver Software Package (Floppy Set)

Once you have developed your driver, you may want to package the driver so that it can be installed on other UNIX Systems. The scheme for putting a DSP on a floppy diskette is the same as the packaging used for UNIX System V/386 software packages you might obtain from AT&T or elsewhere. You must create the files (called modules) that were identified in the section "The Driver Software Package."

To begin, create a directory on your UNIX System V/386, and place the **Driver.o**, **Node**, **Master**, and **System** files there along with the **Init**, **Rc** and **Shutdown** files if they are needed (make sure the **Rc** and **Shutdown** file permissions allow execution by **root**). Also in this directory, place any user commands, programs, or data files to be loaded with your driver. Creating a subdirectory for these items is permissible, but it will be the **Install** script's job to install them. Review the procedures on floppy packaging under **installpkg**. The following additional requirements must be addressed when building an ID floppy set.

The Size File

In addition to the space required to hold any user commands, programs, or data files that your package installs, the system you are installing on will need a large amount of free disk space for the Kernel reconfiguration. See the section "Size (required)."

The Files File

This file should only contain the user programs, commands, and data files installed by your package, not the ID modules.

The Install Script

Appendix E contains a sample script you should review. 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. Perform ID kernel reconfiguration before you install any user commands or data files. This simplifies cleanup if the kernel reconfiguration terminates for some reason.
3. Call **idinstall** and exit appropriately on errors. Use the **echo** and/or **message** commands to tell the user what failed.
4. 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. Install any user programs, data files, or commands included in your package.

The Remove 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. See Appendix E.

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

put a formatted 360K or 1.2M diskette in the floppy drive
cd to where your files are
touch Size file
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.

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. Although the **Size** file provides this number, the user never sees this information.
- Advise the user not to have any background processes running that will
  - ☐ be adversely affected by a system reboot
  - ☐ consume free disk space while a reconfiguration is under way

  For example, running **uucp** during an installation should be avoided.

Converting XENIX System V/386 Device Drivers to UNIX System V/386 Device Drivers

This section presents information on converting XENIX System V/386 device drivers to UNIX System V/386 device drivers.

In UNIX System V/386, the COFF and **x.out** 286 binaries are supported by the **/bin/i286emul** and **/bin/x286emul** user-level emulators. **i286emul** and **x286emul** trap system calls issued by a 286 program and either handle the system calls internally or perform necessary argument conversions before issuing a 386 system call. Therefore, the XENIX System V/386 device driver code that was used to support system calls from a 286 binary is no longer necessary.

In Release 3.2 of UNIX System V/386, the kernel support routines available for device drivers handling 286 system calls [for example, ldtalloc(), ldtfree(), cvtoint(), and cvtoaddr()] are provided as stubs to help facilitate compilation. These stubs will be removed in a future release of UNIX System V/386.

Programmers should keep the following information in mind when converting XENIX System V/386 device drivers to UNIX System V/386 device drivers:

- All XENIX System V/386 include lines that use the form

  #include "../h/<*file*>**"**

  must be changed to

  #include "sys/<*file*>**"**

- The UNIX System V/386 Software Generation System (SGS) does not define the M_I8086, M_I286, or M_I386 symbols. Instead, the i8086, i286, and i386 symbols can be used for native development.
- UNIX System V/386 does not support the **near** and **far** keywords. All references to **near** and **far** should be removed.

- In UNIX System V/386, the **b_paddr** field has been replaced with the **b_un.b_addr** field, which stores an address as a kernel virtual address. In XENIX System V/386, the **b_paddr** field of the *buf* structure stores an address as a physical address. All references to **b_paddr** should be changed to **b_un.b_addr**. Where appropriate, the ktop() macro should be used to convert the address stored in **b_un.b_addr** to a physical address.
- In UNIX System V/386, the **b_blkno** field of the *buf* structure stores block numbers in units of 512 bytes. In XENIX System V/386, **b_blkno** stores blocks in units of 1024 bytes. Be sure to examine and convert all references of **b_blkno** to the units expected by your device driver.
- In UNIX System V/386, all block devices are required to have an xxprint() routine. The following example shows an xxprint() routine for a floppy diskette device driver:

```
flprint (dev, str)
dev_t dev;
char *str;
{
    cmn_err (CE_NOTE, "%s on floppy diskette unit %d, minor %d", str,
            unitbits (dev), minor(dev));
}
```

- All XENIX System V/386 device driver references to the **cmos.h** include file should be changed to **sys/cram.h**.
- The use of the physio() routine in UNIX System V/386 is slightly different than in XENIX System V/386. In UNIX System V/386, the read and write routines first call the phyck() routine to validate the requested transfer; physio() is then called with a pointer to the device driver's xxbreakup() routine. xxbreakup() then calls the system breakup routine [either dma_breakup() or pio_breakup()] with a pointer to the device driver's xxstrategy routine. In XENIX System V/386, a driver's read and write routines called the physio() routine with a pointer to the driver's strategy routine (possibly with **B_TAPE** set).

The following example illustrates the UNIX System calling convention:

```
flbreakup(bp)
struct buf     *bp;
{
int     flstrategy();

        dma_breakup(flstrategy, bp);
}

flread(dev)
dev_t dev;
{
        register int size;

        size = flblktosec(flsize[sizeindx(dev)]);
        /* size in sectors */
        if (physck(size, B_READ))
                physio(flbreakup, NULL, dev, B_READ);
}
```

- In UNIX System V/386, the user structure no longer contains the **u_cpu** field. A new field in the user structure, **u_renv**, contains the same information as **u_cpu** in bits 16-23.
- UNIX System V/386 calls the open, close, read, write, and ioctl routines with the entire device number. XENIX System V/386 calls these routines with the minor device number. When converting XENIX System V/386 device drivers, be sure to mask off the major portion of the device number only if the minor number is desired. This can be done using the minor() macro.
- After all device driver halt routines are called (those defined in the array **io_halt[ ]**), interrupts may be turned on again. If the UNIX System V/386 device driver is used to control hardware, its halt routine should ensure that no interrupt is pending.

- In UNIX System V/386, the disksort() routine uses the **b_sector** field of the *buf* structure to sort requests. In XENIX System V/386, disksort() uses the **b_cylin** field of the *buf* structure to sort requests. By using the **b_sector** field (which is a 32-bit field) better resolution can be obtained over the **b_cylin** field (which is a 16-bit field).
- UNIX System V/386 does not support the XENIX System V/386 mapptov() routine. Instead, the mappages() routine should be used. The mappages() interface is shown below:

```
mappages (begmapaddr, length, begphysaddr)
caddr_t begmapaddr;
int length;
paddr_t begphysaddr;
```

NOTE

Often, XENIX programmers did not use the ktop() macro to convert virtual addresses to physical addresses (for example, in the first call to copyio(), which expects a physical address) because in XENIX physical addresses are equivalent to virtual addresses. In UNIX System V/386, programmers cannot make this assumption; they must use a physical or virtual address where needed using the proper conversion macro where appropriate.

4 Porting

Porting

Portability is the ease in which applications or operating systems can be adapted to run on machines other than the ones they were designed for.

This chapter focuses on techniques and restrictions that should be considered when porting from one machine to another.

Programming Techniques

An application program must be machine-independent to be truly portable. Consequently, when writing a program, do not depend on certain "hooks" within your host machine to execute the program. To make this task easier, the following useful practices consciously isolate any system-inherent hooks into manageable sections:

- When dealing with version-related software releases, isolate any features or functions that are dependent on a specific version of the software.
- Be careful with the manipulation of internal data structures that assume padding or other structural elements.
- Whenever appropriate, try to use only library functions/routines and system calls that are common to the systems.
- When certain "non-portable" elements of the program are required, attempt to define a common version of the feature that can be adapted to the allowable ranges of portability to the systems.
- Make an effort to program in a subset of C that is determined by the commonalities of the compilers of the target systems.
- Determine the formats of data transfers in a hardware-independent fashion.
- As part of good coding practice, use **lint** (if possible) to warn you of any porting problems.

Portability Restrictions

A portable application program cannot rely on any of the following restrictions to operate. Keep in mind, however, that violation of one or more of the restrictions does not necessarily mean that the executable output file will not be portable to any other system. It does mean that there can exist one or more systems where the program will not execute correctly. In the following discussions, these restrictions are elaborated on. Each is followed by a brief discussion on adding new features.

- the type or number of input/output devices provided
- more than one minimum utilities set
- more than the minimum system call set
- more than the minimum amount of available memory space
- tunable system parameters
- use of absolute memory addresses
- more than the specified set of absolute path names
- unrestricted use of system header files
- sign extension

Input/Output Devices

A program is not portable if it depends on specific types or numbers of system interfaces, or special device files. However, programs should access generic special devices and standard input/output facilities.

Utilities Set

A program is not portable if it depends on specific utilities sets not available on the target system. Appendix F provides a list of standard utilities sets.

System Calls

A program depending on system calls other than those available on the target system is not portable. Appendix F provides a list of standard system calls. This is also true of programs relying on certain system signals or command arguments. Appendix F also provides a list of standard system signals and command arguments, respectively.

Memory Space

A program that depends on large amounts of memory space for proper operation rather than relying on secondary storage may not be portable. The physical address space available to a program depends on the amount of memory of a particular system, including the number of drivers and tunable system parameters.

Tunable System Parameters

Any program requiring tunable system parameters to be outside of the ranges specified in the base system **mtune** file (see *mtune*(**4**) in Appendix A) is not portable.

Absolute Memory Addresses

A program that requires absolute addresses or a range of absolute addresses for proper operation is not portable. This includes references to registers or memory mapped input/output devices available through header file definitions. This also applies to shared memory system calls that allow specification of the shared memory space. A program is in violation of this restriction if it performs arithmetic that assumes a specific memory range.

Absolute Path Names

A program is not portable if it assumes absolute path names or files other than those allowed on the target system. A list of standard files are provided in Appendix F. A list of absolute path names is provided below:

/bin /lib /usr/bin
/dev /tmp /usr/tmp
/etc

System Header Files

A program that uses system-specific information by including certain header files may not be portable. Some of these header files are used for system calls. Check before using any of these files.

Sign Extension

When a *char* type is promoted to an *int*, there may or may not be sign extension, depending on the compiler. For example:

```
char c;
int i;

c = '377';
i = c;
```

With some compilers, `i` has a value of 0377 (no sign extension); with others, `i` has a value of -1. This difference can cause all sorts of problems. For example, one often uses characters as indices into an array:

```
char trans[256];
char c, d;


        c = '200';
        trans[c] = d;
```

The index will be negative if `c` > 0177.

You might run into these problems when porting to the 386, which has a sign-extending compiler. For example, this problem surfaces when non-ASCII code sets are used, such as Western European. Those code sets have characters whose codes are > 0177 in addition to ASCII characters.

The solution to this problem is to declare *chars* as unsigned *chars* or add casts. For example:

```
char trans[256];
unsigned char c, d;

        c = '200';
        trans[c] = d;

char trans[256];
char c, d;

        c = '200';
        trans[(unsigned char)c] = d;
```

Other problems that surface are comparisons. For example:

```
char c, d;

if(c < d)
```

Adding New Features

New features must be added carefully to maintain the portability of a program. If absolute portability is required, new features must be anticipated in early releases. If this is not possible, new features must be either disallowed or isolated from the portable systems.

Note that if a new feature is not optional, it may affect compatibility of the entire system. The decision to add this type of feature must be a conscious one.

5 Security

Security Notes

Enhancements to Security

Improvements in security have been made in this release of UNIX System V/386. This chapter provides information about some of these improvements.

Sticky Bit

Public directories like **/tmp**, **/usr/tmp**, and any other directories writable by the world or by a group are vulnerable to the removal of their files by any process. This poses a serious problem to the integrity of files contained in those directories, as well as to the overall security of the system. To avoid potential security problems, a solution is to use the "sticky bit" on a directory to show the restriction on the removal of objects within the directory.

In this release, the sticky bit is set (by default) on the public directories **/tmp** and **/usr/tmp**. Before a user can remove files or directories from these publicly readable and writable directories, some special requirements (given later in this discussion) must be satisfied. Formerly, anyone could remove any file or directory in **/tmp** and **/usr/tmp**.

New in this release is the added functionality of the sticky bit on a directory being settable by a user. In previous releases of UNIX System V/386, the sticky bit could only be set by the super-user. If a directory is writable and the sticky bit is on, a user can only remove a file in that directory if one or more of the following is true:

1. The user owns the file.
2. The user owns the directory.
3. The file is writable by the user.
4. The user is the super-user.

For information on how to set the sticky bit, see *chown*(1) in the *User's/System Administrator's Reference Manual.*

The sticky bit on a directory is set by a regular user via the **chmod** command and the **chmod** system call. The new file deletion semantics will be controlled by the **unlink** and **rmdir** system calls, because all commands that remove files (that is, **mv**, **rm**, and **rmdir**) use those calls to do the removal.

Shadow Password

The shadow password file is being developed to address a major security concern in UNIX System V/386. Currently, encrypted user passwords reside in the password file (**/etc/passwd**), which is readable by all users. Unfriendly remote users can obtain the password file through a command such as **uucp**. They can then either find logins that have no password and use them to access the system or they can try to crack the passwords by using password generation programs. To deter such activities, encrypted passwords and their aging information are being moved to an access-restricted file called the "shadow password" file (**/etc/shadow**). The shadow password file will only be readable by **root**.

The shadow password file contains one entry per login. Each entry consists of the following information:

username the user's login name (ID)

password a 13-character encrypted password for the user or a *lock* string to indicate that the login is not accessible

lastchanged the number of days since January 1, 1970, that the password has been modified

min the minimum number of days required between password changes

max the maximum number of days the password is valid

User Commands

cron

When **cron** completes a job, it sends a mail message to the user. This mail message had been sent by **root** but will now be sent with the ID of the user who initiated the cron job.

login

In the past, 10 login attempts were permitted before the line would be dropped. That limit is now changed to 5 attempts. After the fifth unsuccessful login attempt, **login** will sleep for 20 seconds before dropping the line.

mail

The following special characters are not valid in the mail forwarding line:

```
; & ¦ < > ' ` " ? * [ ] { } ( ) $ #    \
```

If a special character is encountered in the "forward to" line, the mail will be returned to the sender with the message

```
invalid address
```

uucp

System names must not contain unprintable characters or any of these special characters:

```
; & ¦ < > ' ` " ? * [ ] { } ( ) $ #    \
```

The **uuxqt** daemon will not perform remote execution requests for systems whose names contain any of these characters.

getspent

The **getspent** routine when first called returns a pointer to the first *spwd* structure in the *shadow* file. Refer to Appendix A for the manual page on this routine.

putspent

The **putspent** routine is the inverse of **getspent**. It writes a line on the stream *fp*, which matches the format of **/etc/shadow**. Refer to Appendix A for the manual page on this routine.

System Administrator Commands

cu

A new entry called **uudirect** is defined in **/usr/lib/uucp/Devices**. When doing a **cu** from machine A to machine B on a direct line where **uugetty** is running on that line on the remote machine, an extra carriage return must be sent so that **uugetty** knows whether it is an incoming or outgoing line. In the **/usr/lib/uucp/Systems** file, **uudirect** should be used for the *devices* field. For more information on this file, refer to *uucp*(1C) in the *User's/System Administrator's Reference Manual.*

loginlog

To enhance **login** security, the time of the last login will be displayed after logging in. To turn on the mechanism that logs unsuccessful attempts to access the system, the administrator must create the file **/usr/adm/loginlog**. If the file **loginlog** exists and five consecutive unsuccessful login attempts occur, all will be logged in **loginlog**, then **login** will sleep for 20 seconds before dropping the line. In other words, if a person tries five times unsuccessfully to log in at a terminal, all five attempts will be logged in **/usr/adm/loginlog** if the file exists. The **login** component will then sleep for 20 seconds and drop the line. If a person has one or two unsuccessful attempts, none of them will be logged.

To enable logging, the log file should be created with read and write permission for owner only. It should be owned by **root** with group **sys**.

The **/usr/adm/loginlog** file is a text file. It will contain one entry for each unsuccessful attempt. Entries in **/usr/adm/loginlog** will have the following format:

login name:tty specification:time

The *login name* field contains the login name used in the failed login attempt. The *tty specification* field contains the terminal location of the login attempt, and *time* contains the approximate time of the login attempt. You must create this file and turn logging on.

lp Commands

The **disable** command can be made non-executable for regular users by logging in as **root** or **lp** and executing

chmod 4550 /usr/bin/disable

The **cancel** command can be made non-executable for regular users by logging in as **root** or **lp** and executing

chmod 4550 /usr/bin/cancel

pwconv

If you attempt to run **pwconv** with an incorrectly formatted line on **/etc/passwd** (for example, one with too many colons), **pwconv** will stop scanning at the bad line. It will complete the conversion, but the new password file and the **/etc/shadow** file will both have too few lines. You will still be able to recover the original **/etc/passwd** by restoring the old file stored as **/etc/opasswd**. To restore the old password file, follow these directions:

Step 1: Replace the new password file with the original one:

mv /etc/opasswd /etc/passwd

Step 2: Remove the shadow password file:

rm /etc/shadow

Step 3: Find and correct the bad entries in **/etc/password** using **vi** or **ed.**

The recommended way of changing the password file is with the tools provided, not with the editors. However, because the only way to corrupt a password file is by using an editor, that becomes the only way to correct the error as well. The use of any of the editors is still not recommended.

Step 4: Run **/usr/bin/pwconv** again.

/usr/lib/uucp/remote.unknown

The **remote.unknown** file has been changed from a shell script to a C language executable program. **remote.unknown** is part of the Basic Networking Utilities delivered in the Base System Package of the UNIX System V/386 Foundation Set. When an unknown machine starts a conversation with the local machine, **remote.unknown** logs the conversation attempt.

/usr/spool/cron

The **/usr/spool/cron** directory, which contains directories for **at**, **cron**, and **crontab** jobs, will no longer be accessible by users. The directory mode will now be set to 700.

/usr/spool/uucp

Except for **root**, users will no longer be able to write in the **/usr/spool/uucp** directory or any directories under it. The directory mode will now be set to 755.

Compatibility Notes

Shell Scripts

It is strongly recommended that all applications convert any shell scripts into binary programs if specific user (group) permissions are required in the shell script command lines. To pass permissions, the binary program must have the setuid (gid) mode bit on and the owner (group) of the binary program set to the ID required. Then the **exec** system call can be invoked with the binary program as the argument and the correct permissions will be passed.

If it is not possible to convert the shell scripts into binary programs, then a binary interface program must be written that would have the setuid mode bit on and the owner of the file set to **root**. Next, the process would have to do a **setuid** (GID) system call internally with the UID (GID) that must be passed to a sub-shell. This is only possible because the **setuid** (**setgid**) system call sets both the real and effective UID (GID) when called by a process with the effective UID of **root**. Finally, the binary interface would then call the shell script. This is a potentially dangerous procedure unless the programmer is aware of

all the implications.

PATH

The default **PATH** environment variable searches the current directory first. A super-user unknowingly may run a program in the current directory. The super-user should change the **ROOTPATH** so that the current directory is searched last instead of first.

login

To discourage intruders, the encrypted password and password aging information formerly found in **/etc/passwd** has been moved to **/etc/shadow**. This file can only be read by the super-user. Users will still be able to change their passwords using the **passwd** command. Password and aging information is added to **/etc/shadow** by running a new program, **pwconv**. This program can only be executed by the super-user.

If you have an application or program that writes password and/or aging information into **/etc/passwd**, the program will have to be modified so that **pwconv** is executed after the information is appended to **/etc/passwd**. Until the modification can be made, the administrator with super-user privilege will have to run the program before the user who has been added or whose password information has been modified can log in.

ps

In this release of UNIX System V/386, the usage of **ps** has changed. Now **ps** checks and sets the user's effective UID to the real UID and the effective GID to the real GID. Therefore, only users with a real user ID of **root** or a real group ID of **sys** will be able to use these options to **ps**.

edit, ex, vedit, vi, view

The **edit**, **ex**, **vedit**, **vi**, and **view** commands allow separate **.exrc** files in any directory. In addition, if you change directory to another user's directory and use any of these editors to edit a file in that other user's directory, the editor will execute the **.exrc** file if it exists in the second user's directory. This functionality has security implications depending on the contents of the **.exrc** file, because the commands are executed as the user invoking the editor and not as the person who owns the **.exrc** file.

In this release, a new option has been added to the **vi** and **ex** commands to allow you the option of reading the **.exrc** file in the current directory. Initially, the flag is not set, that is, the **vi/ex** command will NOT read the **.exrc** file if it exists in the current working directory. You can modify this option by inserting the line

```
set exrc
```

or the abbreviation

```
set ex
```

in the **$HOME/.exrc** file, which is read when one of these editors is executed if the **EXINIT** variable is not set in the **.profile**. If you want to set the **EXINIT** variable, add the following lines to your **.profile**:

```
EXINIT="set exrc"
export EXINIT
```

However, you should note that executing **vi** or **ex** as another user with **su** could result in your files being compromised, since certain variables in the environment are passed when **su** is executed without the **"-"** option.

For more information, see the *ex*(1) manual page in the *User's/System Administrator's Reference Manual*.

A Appendix A

Manual Pages

This appendix contains manual pages for those ID programs and files that a device driver writer needs to know about; that is, programs and files that may be needed by the Install or Remove script, or that may be used during driver development. It also contains manual pages for security user commands.

- *idbuild*(1M) - A shell script that does the complete system reconfiguration.
- *idcheck*(1M) - A command that tests for the presence of a driver or checks the availability of a particular IVN, IOA, or CMA.
- *idconfig*(1M) - A command that produces a new kernel configuration.
- *idinstall*(1M) - A command that installs, updates, or removes a driver package or driver component.
- *idmkinit*(1M) - A command that updates **/etc/inittab**.
- *idmknod*(1M) - A command that updates device nodes in the **/dev** directory.
- *idmkunix*(1M) - A command that builds a new UNIX Operating System kernel.
- *idspace*(1M) - A command that interrogates free space in one or more file systems.
- *idtune*(1M) - A shell script that specifies system tunable parameters.
- *getspent*(3X) - A command that gets shadow password file entry.
- *putspent*(3X) - A command that writes shadow password file entry.
- *mdevice*(4) - The device master file (and DSP *Master* component format).
- *mfsys*(4) - The file system type master data.
- *mtune*(4) - The tunable parameter master file.
- *sdevice*(4) - The device system file (and DSP *System* component format).

- *sfsys*(4) - File system type system data.
- *stune*(4) - A tunable parameter system file.

NAME

idbuild – build new UNIX System kernel

SYNOPSIS

/etc/conf/bin/idbuild

DESCRIPTION

This script builds a new UNIX System kernel using the current system configuration in **etc/conf/**. Kernel reconfigurations are usually done after a device driver is installed, or system tunable parameters are modified. The script uses the shell variable **$ROOT** from the user's environment as its starting path. Except for the special case of kernel development in a non-root source tree, the shell variable **ROOT** should always be set to null or to "/". *idbuild* exits with a return code of zero on success and non-zero on failure.

Building a new UNIX System image consists of generating new system configuration files, then link-editing the kernel and device driver object modules in the **etc/conf/pack.d** object tree. This is done by *idbuild* by calling the following commands:

etc/conf/bin/idconfig To build kernel configuration files.

etc/conf/bin/idmkunix To process the configuration files and link-edit a new UNIX System image.

The system configuration files are built by processing the Master and System files representing device driver and tunable parameter specifications. For the i386 UNIX System the files **etc/conf/cf.d/mdevice,** and **etc/conf/cf.d/mtune** represent the Master information. The files **etc/conf/cf.d/stune,** and the files specified in **etc/conf/sdevice.d/*** represent the System information. The kernel also has file system type information defined in the files specified by **etc/conf/sfsys.d/*** and **etc/conf/mfsys.d/***.

Once a new UNIX System kernel has been configured, a lock file is set in **etc/.new_unix** which causes the new kernel to replace **/unix** on the next system shutdown (i.e., on the next entry to the *init 0* state). Upon the next system boot, the new kernel will be executed.

ERROR MESSAGES

Since *idbuild* calls other system commands to accomplish system reconfiguration and link editing, it will report all errors encountered by those commands, then clean up intermediate files created in the process. In general, the exit value 1 indicates an error was encountered by *idbuild*.

The errors encountered fall into the following categories:

Master file error messages.
System file error messages.
Tunable file error messages.
Compiler and Link-editor error messages.

All error messages are designed to be self-explanatory.

SEE ALSO

idinstall(1m), idtune(1m).

mdevice(4), mfsys(4), mtune(4), sdevice(4), sfsys(4), stune(4) in the *Programmer's Reference Manual.*

NAME

idcheck – returns selected information

SYNOPSIS

/etc/conf/bin/idcheck

DESCRIPTION

This command returns selected information about the system configuration. It is useful in add-on device Driver Software Package (DSP) installation scripts to determine if a particular device driver has already been installed, or to verify that a particular interrupt vector, I/O address or other selectable parameter is in fact available for use. The various forms are:

idcheck –p device-name [–i dir] [–r]

idcheck –v vector [–i dir] [–r]

idcheck –d dma-channel [–i dir] [–r]

idcheck –a –l lower_address –u upper_address [–i dir] [–r]

idcheck –c –l lower_address –u upper_address [–i dir] [–r]

This command scans the System and Master modules and returns:

100 if an error occurs.

0 if no conflict exists.

a positive number greater than 0 and less than 100 if a conflict exists.

The command line options are:

–r Report device name of any conflicting device on stdout.

–p *device-name* This option checks for the existence of four different components of the DSP. The exit code is the addition of the return codes from the four checks.

Add 1 to the exit code if the DSP directory under **/etc/conf/pack.d** exists.

Add 2 to the exit code if the Master module has been installed.

Add 4 to the exit code if the System module has been installed.

Add 8 to the exit code if the Kernel was built with the System module.

Add 16 to the exit code if a Driver.o is part of the DSP (vs. a stubs.c file).

–v *vector* Returns 'type' field of device that is using the vector specified (i.e., another DSP is already using the vector).

–d *dma-channel* Returns 1 if the dma channel specified is being used.

–a This option checks whether the IOA region bounded by "lower" and "upper" conflict with another DSP

("lower" and "upper" are specified with the **-l** and **-u** options). The exit code is the addition of two different return codes.

Add 1 to the exit code if the IOA region overlaps with another device.

Add 2 to the exit code if the IOA region overlaps with another device and that device has the 'O' option specified in the *type* field of the Master module. The 'O' option permits a driver to overlap the IOA region of another driver.

-c Returns 1 if the CMA region bounded by "lower" and "upper" conflict with another DSP ("lower" and "upper" are specified with the **-l** and **-u** options).

-l *address* Lower bound of address range specified in hex. The leading 0x is unnecessary.

-u *address* Upper bound of address range specified in hex. The leading 0x is unnecessary.

-i *dir* Specifies the directory in which the ID files *sdevice* and *mdevice* reside. The default directory is **/etc/conf/cf.d**.

ERROR MESSAGES

There are no error messages or checks for valid arguments to options. *idcheck* interprets these arguments using the rules of *scanf*(3) and queries the *sdevice* and *mdevice* files. For example, if a letter is used in the place of a digit, *scanf*(3) will translate the letter to 0. *idcheck* will then use this value in its query.

SEE ALSO

idinstall(1m).

mdevice(4), sdevice(4) in the *Programmer's Reference Manual.*

NAME

idconfig – produce a new kernel configuration

SYNOPSIS

/etc/conf/bin/idconfig

DESCRIPTION

The *idconfig* command takes as its input a collection of files specifying the configuration of the next UNIX System to be built. A collection of output files for use by *idmkunix* is produced.

The input files expected by *idconfig* are as follows:

mdevice – Master device specifications
sdevice – System device specifications
mtune – Master parameter specifications
stune – System parameter specifications
mfsys – File system type master data
sfsys – File system type system data
sassign – Device Assignment File

The output files produced by *idconfig* are as follows:

conf.c – Kernel data structures and function definitions
config.h – Kernel parameter and device definitions
vector.c – Interrupt vector definitions
direct – Listing of all driver components included in the build
fsconf.c – File system type configuration data

The command line options are as follows:

-o *directory* Output files will be created in the directory specified rather than **/etc/conf/cf.d**.

-i *directory* Input files that normally reside in **/etc/conf/cf.d** can be found in the directory specified.

-r *directory* The directory specified will be used as the ID "root" directory rather than **/etc/conf**.

-d *file* Use *file* name rather than *sdevice* for input.

-t *file* Use *file* name rather than *stune* for input.

-T *file* Use *file* name rather than *mtune* for input.

-a *file* Use *file* name rather than *sassign* for input.

-c *file* Redirect *conf.c* output to *file* name.

-h *file* Redirect *config.h* output to *file* name.

-v *file* Redirect *vector.c* output to *file* name.

-p *file* Redirect *direct* output to *file* name.

-D, -m, -s These options are no longer supported.

-# Print debugging information.

ERROR MESSAGES

An exit value of zero indicates success. If an error was encountered, *idconfig* will exit with a non-zero value and report an error message. All error messages are designed to be self-explanatory.

SEE ALSO

dmkunix(1M), idbuild(1M), idinstall(1M), mdevice(4), mtune(4), sdevice(4), stune(4).

NAME

idinstall – add, delete, update, or get device driver configuration data

SYNOPSIS

/etc/conf/bin/idinstall –[**adug**] [-**e**] –[**msoptnirhcl**] dev_name

DESCRIPTION

The *idinstall* command is called by a Driver Software Package (DSP) Install script or Remove script to Add (**-a**), Delete (**-d**), Update (**-u**), or Get (**-g**) device driver configuration data. *idinstall* expects to find driver component files in the current directory. When components are installed or updated, they are moved or appended to files in the **/etc/conf** directory and then deleted from the current directory unless the *-k* flag is used. The options for the command are as follows:

Action Specifiers:

- **-a** Add the DSP components
- **-d** Remove the DSP components
- **-u** Update the DSP components
- **-g** Get the DSP components (print to std out, except Master)

Component Specifiers: (*)

- **-m** Master component
- **-s** System component
- **-o** Driver.o component
- **-p** Space.c component
- **-t** Stubs.c component
- **-n** Node (special file) component
- **-i** Inittab component
- **-r** Device Initialization (rc) component
- **-h** Device shutdown (sd) component
- **-c** Mfsys component: file system type config (Master) data
- **-l** Sfsys component: file system type local (System) data

(*) If no component is specified, the default is all except for the **-g** option where a single component must be specified explicitly.

Miscellaneous:

- **-e** Disable free disk space check
- **-k** Keep files (do not remove from current directory) on add or update.

In the simplest case of installing a new DSP, the command syntax used by the DSP's Install script should be **idinstall -a** *dev_name*. In this case the command will require and install a Driver.o, Master and System entry, and

optionally install the Space.c, Stubs.c, Node, Init, Rc, Shutdown, Mfsys, and Sfsys components if those modules are present in the current directory.

The Driver.o, Space.c, and Stubs.c files are moved to a directory in **/etc/conf/pack.d.** The *dev_name* is passed as an argument, which is used as the directory name. The remaining components are stored in the corresponding directories under **/etc/conf** in a file whose name is *dev_name*. For example, the Node file would be moved to **/etc/conf/node.d/dev_name.**

The *idinstall -m* usage provides an interface to the *idmaster* command which will add, delete, and update *mdevice* file entries using a Master file from the local directory. An interface is provided here so that driver writers have a consistent interface to install any DSP component.

As stated above, driver writers will generally use only the **idinstall -a** *dev_name* form of the command. Other options of *idinstall* are provided to allow an Update DSP (i.e., one that replaces an existing device driver component) to be installed, and to support installation of multiple controller boards of the same type.

If the call to *idinstall* uses the **-u** (update) option, it will:

overlay the files of the old DSP with the files of the new DSP.

invoke the *idmaster* command with the 'update' option if a Master module is part of the new DSP.

idinstall also does a verification that enough free disk space is available to start the reconfiguration process. This is done by calling the *idspace* command. *idinstall* will fail if insufficient space exists, and exit with a non-zero return code. The **-e** option bypasses this check.

idinstall makes a record of the last device installed in a file **(/etc/.last_dev_add) ,** and saves all removed files from the last delete operation in a directory **(/etc/.last_dev_del).** These files are recovered by */etc/conf/bin/idmkenv* whenever it is determined that a system reconfiguration was aborted due to a power failure or unexpected system reboot.

ERROR MESSAGES

An exit value of zero indicates success. If an error was encountered, *idinstall* will exit with a non-zero value, and report an error message. All error messages are designed to be self-explanatory. Typical error message that can be generated by *idinstall* are as follows:

Device package already exists.
Cannot make the driver package directory.
Cannot remove driver package directory.
Local directory does not contain a Driver object (Driver.o) file.
Local directory does not contain a Master file.
Local directory does not contain a System file.
Cannot remove driver entry.

SEE ALSO

idspace(1m), idcheck(1m).
mdevice(4), sdevice(4) in the *Programmer's Reference Manual.*

NAME

idmkinit – reads files containing specifications

SYNOPSIS

/etc/conf/bin/idmkinit

DESCRIPTION

This command reads the files containing specifications of **/etc/inittab** entries from **/etc/conf/init.d** and constructs a new **inittab** file in **/etc/conf/cf.d**. It returns 0 on success and a positive number on error.

The files in **/etc/conf/init.d** are copies of the Init modules in device Driver Software Packages (DSP). There is at most one Init file per DSP. Each file contains one line for each **inittab** entry to be installed. There may be multiple lines (i.e., multiple **inittab** entries) per file. An **inittab** entry has the form (the *id* field is often called the *tag*):

id:rstate:action:process

The Init module entry must have one of the following forms:

action:process

rstate:action:process

id:rstate:action:process

When *idmkinit* encounters an entry of the first type, a valid **id** field will be generated, and an **rstate** field of 2 (indicating run on init state 2) will be generated. When an entry of the second type is encountered only the **id** field is prepended. An entry of the third type is incorporated into the new **inittab** unchanged.

Since add-on **inittab** entries specify init state 2 for their **rstate** field most often, an entry of the first type should almost always be used. An entry of the second type may be specified if you need to specify other than state 2. DSP's should avoid specifying the **id** field as in the third entry, since other add-on applications or DSPs may have already used the **id** value you have chosen. The */etc/init* program will encounter serious errors if one or more **inittab** entries contain the same **id** field.

Idmkinit determines which of the three forms above is being used for the entry by requiring each entry to have a valid **action** keyword. Valid **action** values are as follows:

off
respawn
ondemand
once
wait
boot
bootwait
powerfail
powerwait
initdefault
sysinit

The *idmkinit* command is called automatically upon entering init State 2 on the next system reboot after a kernel reconfiguration to establish the correct **/etc/inittab** for the running **/unix** kernel. *idmkinit* can be called as a user level command to test modification of **inittab** before a DSP is actually built. It is also useful in installation scripts that do not reconfigure the kernel, but need to create **inittab** entries. In this case, the **inittab** generated by *idmkinit* must be copied to **/etc/inittab**, and a *telinit q* command must be run to make the new entry take affect.

The command line options are:

-o *directory* **inittab** will be created in the directory specified rather than **/etc/conf/cf.d**.

-i *directory* The ID file **init.base**, which normally resides in **/etc/conf/cf.d**, can be found in the directory specified.

-e *directory* The Init modules that are usually in **/etc/conf/init.d** can be found in the directory specified.

-# Print debugging information.

ERROR MESSAGES

An exit value of zero indicates success. If an error was encountered, *idmkinit* will exit with a non-zero value and report an error message. All error messages are designed to be self-explanatory.

SEE ALSO

idbuild(1), idinstall(1m), idmknod(1m), init(1m).

inittab(4) in the *Programmer's Reference Manual.*

NAME

idmknod – removes nodes and reads specifications of nodes

SYNOPSIS

/etc/conf/bin/idmknod

DESCRIPTION

This command performs the following functions:

Removes the nodes for non-required devices (those that do not have an 'r' in field 3 of the the device's *mdevice* entry) from **/dev**. Ordinary files will not be removed. If the **/dev** directory contains subdirectories, those subdirectories will be transversed and nodes found for non-required devices will be removed as well. If empty subdirectories result due to the removal of nodes, the subdirectories are then removed.

Reads the specifications of nodes given in the files contained in **/etc/conf/node.d** and installs these nodes in **/dev**. If the node specification defines a path containing subdirectories, the subdirectories will be made automatically.

Returns 0 on success and a positive number on error.

The *idmknod* command is run automatically upon entering init state 2 on the next system reboot after a kernel reconfiguration to establish the correct representation of device nodes in the **/dev** directory for the running **/unix** kernel. *idmknod* can be called as a user level command to test modification of the **/dev** directory before a DSP is actually built. It is also useful in installation scripts that do not reconfigure the kernel, but need to create **/dev** entries.

The files in **/etc/conf/node.d** are copies of the *Node* modules installed by device Driver Software Packages (DSP). There is at most one file per DSP. Each file contains one line for each node that is to be installed. The format of each line is:

Name of device entry (field 1) in the *mdevice* file (The *mdevice* entry will be the line installed by the DSP from its *Master* module). This field must be from 1 to 8 characters in length. The first character must be a letter. The others may be letters, digits, or underscores.

Name of node to be inserted in **/dev.** The first character must be a letter. The others may be letters, digits, or underscores. This field can be a path relative to **/dev**, and *idmknod* will create subdirectories as needed.

The character **b** or **c**. A **b** indicates that the node is a 'block' type device and **c** indicates 'character' type device.

Minor device number. This value must be between 0 and 255. If this field is a non-numeric, it is assumed to be a request for a streams clone device node, and *idmknod* will set the minor number to the value of the major number of the device specified.

Some example node file entries are as follows:

asy tty00 c 1 makes /dev/tty00 for device 'asy' using minor device 1.

qt rmt/c0s0 c 4 makes /dev/rmt/c0s0 for device 'qt' using minor device 4.

clone net/nau/clone c nau
makes /dev/net/nau/clone for device 'clone'. The minor device number is set to the major device number of device 'nau'.

The command line options are:

-o *directory* Nodes will be installed in the directory specified rather than **/dev**.

-i *directory* The file *mdevice* which normally resides in **/etc/conf/cf.d**, can be found in the directory specified.

-e *directory* The *Node* modules that normally reside in **/etc/conf/node.d** can be found in the directory specified.

-s Suppress removing nodes (just add new nodes).

ERROR MESSAGES

An exit value of zero indicates success. If an error was encountered due to a syntax or format error in a *node* entry, an advisory message will be printed to *stdout* and the command will continue. If a serious error is encountered (i.e., a required file cannot be found), *idmknod* will exit with a non-zero value and report an error message. All error messages are designed to be self-explanatory.

SEE ALSO

idinstall(1m), idmkinit(1m).

mdevice(4), sdevice(4) in the *Programmer's Reference Manual*.

NAME

idmkunix – build new UNIX System kernel

SYNOPSIS

/etc/conf/bin/idmkunix

DESCRIPTION

The *idmkunix* command creates a bootable UNIX Operating System kernel in the directory **/etc/conf/cf.d**. The component kernel "core" files and device driver object files contained in subdirectories of **/etc/conf/pack.d** are used as input along with device and parameter definition files produced by *idconfig*. In brief, the required input files are as follows:

/etc/conf/cf.d/conf.c – Kernel data structures and function definitions
/etc/conf/cf.d/config.h – Kernel parameter and device definitions
/etc/conf/cf.d/vector.c – Interrupt vector definitions
/etc/conf/cf.d/direct – Listing of all driver components included in the build
/etc/conf/cf.d/fsconf.c – File system type configuration data
/etc/conf/cf.d/vuifile – Memory management definitions for the kernel
/etc/conf/pack.d/*/Driver.o – Component kernel object files
/etc/conf/pack.d/*/space.c – Component kernel space allocation files
/etc/conf/pack.d/*/stubs.c – Component kernel stubs files

The command line options are as follows:

-o *directory* The file **unix** be created in the directory specified rather than **/etc/conf/cf.d.**

-i *directory* Input files that normally reside in **/etc/conf/cf.d** can be found in the directory specified.

-r *directory* The directory specified will be used as the ID "root" directory rather than **/etc/conf**.

-c, cc, -l, ld These options are no longer supported.

-# Print debugging information.

ERROR MESSAGES

An exit value of zero indicates success. If an error was encountered, *idmkunix* will exit with a non-zero value and report an error message. All error messages are designed to be self-explanatory.

SEE ALSO

idbuild(1M), idconfig(1M), idinstall(1M), mdevice(4), mtune(4), sdevice(4), stune(4).

NAME

idspace – investigates free space

SYNOPSIS

/etc/conf/bin/idspace [**-i** inodes] [**-r** blocks] [**-u** blocks] [**-t** blocks]

DESCRIPTION

This command investigates free space in **/**, **/usr**, and **/tmp** file systems to determine whether sufficient disk blocks and inodes exist in each of potentially 3 file systems. The default tests that *idspace* performs are as follows:

Verify that the root file system (/) has 400 blocks more than the size of the current **/unix**. This verifies that a device driver being added to the current **/unix** can be built and placed in the root directory. A check is also made to insure that 100 inodes exist in the root directory.

Determine whether a **/usr** file system exists. If it does exist, a test is made that 400 free blocks and 100 inodes are available in that file system. If the file system does not exist, *idspace* does not complain since files created in **/usr** by the reconfiguration process will be created in the root file system and space requirements are covered by the test in (1.) above.

Determine whether a **/tmp** file system exists. If it does exist, a test is made that 400 free blocks and 100 inodes are available in that file system. If the file system does not exist, *idspace* does not complain since files created in **/tmp** by the reconfiguration process will be created in the root file system and space requirements are covered by the test in (1.) above.

The command line options are:

-i *inodes* This option overrides the default test for 100 inode in all of the *idspace* checks.

-r *blocks* This option overrides the default test for */unix* size + 400 blocks when checking the root (/) file system. When the **-r** option is used, the **/usr** and **/tmp** file systems are not tested unless explicitly specified.

-u *blocks* This option overrides the default test for 400 blocks when checking the **/usr** file system. When the **-u** option is used, the root (/) and **/tmp** file systems are not tested unless explicitly specified. If **/usr** is not a separate file system, an error is reported.

-t *blocks* This option overrides the default test for 400 blocks when checking the **/tmp** file system. When the **-t** option is used, the root (/) and **/usr** file systems are not tested unless explicitly specified. If **/tmp** is not a separate file system, an error is reported.

ERROR MESSAGES

An exit value of zero indicates success. If insufficient space exists in a file system or an error was encountered due to a syntax or format error, *idspace* will report a message. All error messages are designed to be self-explanatory. The specific exit values are as follows:

0 success.

1 command syntax error, or needed file does not exist.

2 file system has insufficient space or inodes.

3 requested file system does not exist (**-u** and **-t** options only).

SEE ALSO

idbuild(1m), idinstall(1m).

NAME

idtune – attempts to set value of a tunable parameter

SYNOPSIS

/etc/conf/bin/idtune [**-f** ¦ **-m**] name value

DESCRIPTION

This script attempts to set the value of a tunable parameter. The tunable parameter to be changed is indicated by *name*. The desired value for the tunable parameter is *value*.

If there is already a value for this parameter (in the **stune** file), the user will normally be asked to confirm the change with the following message:

> Tunable Parameter *name* is currently set to *old_value*.
> Is it OK to change it to *value*? (y/n)

If the user answers **y**, the change will be made. Otherwise, the tunable parameter will not be changed, and the following message will be displayed:

> *name* left at *old_value*.

However, if the **-f** (force) option is used, the change will always be made and no messages will ever be given.

If the **-m** (minimum) option is used and there is an existing value which is greater than the desired value, no change will be made and no message will be given.

If system tunable parameters are being modified as part of a device driver or application add-on package, it may not be desirable to prompt the user with the above question. The add-on package Install script may chose to override the existing value using the **-f** or **-m** options. However, care must be taken not to invalidate a tunable parameter modified earlier by the user or another add-on package.

In order for the change in parameter to become effective, the UNIX System kernel must be rebuilt and the system rebooted.

DIAGNOSTICS

The exit status will ne non-zero if errors are encountered.

SEE ALSO

idbuild(1).

mtune(4), stune(4) in the *Programmer's Reference Manual*.

NAME

getspent, getspnam, setspent, endspent, fgetspent – get shadow password file entry

SYNOPSIS

```
#include <shadow.h>

struct spwd *getspent ( )

struct spwd *getspnam (name)
char *name;

void setspent ( )

void endspent ( )

struct spwd *fgetspent (fp)
FILE *fp;
```

DESCRIPTION

The *getspent* and *getspnam* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/shadow** file. Each line in the file contains a "shadow password" structure, declared in the **<shadow.h>** header file:

```
struct spwd{
        char    *sp_namp;
        char    *sp_pwdp;
        long    sp_lstchg;
        long    sp_min;
        long    sp_max;
};
```

The *getspent* routine, when first called, returns a pointer to the first *spwd* structure in the file; thereafter, it returns a pointer to the next *spwd* structure in the file; so successive calls can be used to search the entire file. The *getspnam* routine searches from the beginning of the file until a login name matching *name* is found and returns a pointer to the particular structure in which it was found. The *getspent* and *getspnam* routines populate the **sp_min** or **sp_max** field with –1 if the corresponding field in **/etc/shadow** is empty. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to the *setspent* routine has the effect of rewinding the shadow password file to allow repeated searches. The *endspent* routine may be called to close the shadow password file when processing is complete.

The *fgetspent* routine returns a pointer to the next *spwd* structure in the stream *fp*, which matches the format of **/etc/shadow**.

FILES

/etc/shadow

SEE ALSO

putspent(3X).

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program will increase more than might be expected.

This routine is for internal use only, compatibility is not guaranteed.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

putspent – write shadow password file entry

SYNOPSIS

```
#include <shadow.h>

int putspent (p, fp)
struct spwd *p;
FILE *fp;
```

DESCRIPTION

The *putspent* routine is the inverse of *getspent*(3X). Given a pointer to a *spwd* structure created by the *getspent* routine (or the *getspnam* routine), the *putspent* routine writes a line on the stream *fp*, which matches the format of **/etc/shadow**.

If the **sp_min** or **sp_max** field of the *spwd* structure is –1, the corresponding **/etc/shadow** field is cleared.

SEE ALSO

getspent(3X).

DIAGNOSTICS

The *putspent* routine returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program will increase more than might be expected.

This routine is for internal use only; compatibility is not guaranteed.

NAME

mdevice – file format.

SYNOPSIS

mdevice

DESCRIPTION

The *mdevice* file is included in the directory */etc/conf/cf.d*. It includes a one-line description of each device driver and configurable software module in the system to be built [except for file system types, see *mfsys*(4)]. Each line in *mdevice* represents the *Master* file component from a Driver Software Package (DSP) either delivered with the base system or installed later via *idinstall*.

Each line contains several whitespace-separated fields; they are described below. Each field must be supplied with a value or a '–' (dash).

1. *Device name*: This field is the internal name of the device or module, and may be up to 8 characters long. The first character of the name must be an alphabetic character; the others may be letters, digits, or underscores.

2. *Function list*: This field is a string of characters that identify driver functions that are present. Using one of the characters below requires the driver to have an entry point (function) of the type indicated. If no functions in the following list are supplied, the field should contain a dash.

   o – open routine

   c – close routine

   r – read routine

   w – write routine

   i – ioctl routine

   s – startup routine

   x – exit routine

   f – fork routine

   e – exec routine

   I – init routine

   h – halt routine

   p – poll routine

   E – kenter routine

   X – kexit routine

   Note that if the device is a 'block' type device (see field 3. below), a *strategy* routine and a *print* routine are required by default.

3. *Characteristics of driver*: This field contains a set of characters that indicate the characteristics of the driver. If none of the characters below apply, the field should contain a dash. The legal characters for

this field are:

i - The device driver is installable.

c - The device is a 'character' device.

b - The device is a 'block' device.

t - The device is a tty.

o - This device may have only one *sdevice* entry.

r - This device is required in all configurations of the Kernel. This option is intended for drivers delivered with the base system only. Device nodes (special files in the */dev* directory), once made for this device, are never removed. See *idmknod*.

S - This device driver is a STREAMS module.

H - This device driver controls hardware. This option distinguishes drivers that support hardware from those that are entirely software (pseudo-devices).

G - This device does not use an interrupt though an interrupt is specified in the *sdevice* entry. This is used when you wish to associate a device to a specific device group.

D - This option indicates that the device driver can share its DMA channel.

O - This option indicates that the IOA range of this device may overlap that of another device.

4. *Handler prefix*: This field contains the character string prepended to all the externally-known handler routines associated with this driver. The string may be up to 4 characters long.

5. *Block Major number*: This field should be set to zero in a DSP *Master* file. If the device is a 'block' type device, a value will be assigned by *idinstall* during installation.

6. *Character Major number*: This field should be set to zero in a DSP *Master* file. If the device is a 'character' type device (or 'STREAMS' type), a value will be assigned by *idinstall* during installation.

7. *Minimum units*: This field is an integer specifying the minimum number of these devices that can be specified in the *sdevice* file.

8. *Maximum units*: This field specifies the maximum number of these devices that may be specified in the *sdevice* file. It contains an integer.

9. *DMA channel*: This field contains an integer that specifies the DMA channel to be used by this device. If the device does not use DMA, place a '-1' in this field. Note that more than one device can share a DMA channel (previously disallowed).

SPECIFYING STREAMS DEVICES AND MODULES

STREAMS modules and drivers are treated in a slightly different way from other drivers in all UNIX Systems, and their configuration reflects this difference. To specify a STREAMS device driver, its *mdevice* entry should contain both an 'S' and a 'c' in the *characteristics* field (see 3. above). This indicates that it is a STREAMS driver and that it requires an entry in the UNIX kernel's *cdevsw* table, where STREAMS drivers are normally configured into the system.

A STREAMS module that is not a device driver, such as a line discipline module, requires an 'S' in the *characteristics* field of its *mdevice* file entry, but should not include a 'c', as a device driver does.

SEE ALSO

mfsys(4), **sdevice(4)**.

idinstall(1m) in the *User's/System Administrator's Reference Manual*.

NAME

mfsys – file format.

SYNOPSIS

mfsys

DESCRIPTION

The *mfsys* file contains configuration information for file system types that are to be included in the next system kernel to be built. It is included in the directory */etc/conf/cf.d*, and includes a one-line description of each file system type. The *mfsys* file is coalesced from component files in the directory */etc/conf/mfsys.d*. Each line contains the following whitespace-separated fields:

1. *name*: This field contains the internal name for the file system type (e.g., S51K, DUFST). This name is no more than 32 characters long, and by convention is composed of upper-case alphanumeric characters.
2. *prefix*: The *prefix* in this field is the string prepended to the *fstypsw* handler functions defined for this file system type (e.g., s5, du). The prefix must be no more that 8 characters long.
3. *flags*: The *flags* field contains a hex number of the form "0xNN" to be used in populating the *fsinfo* data structure table entry for this file system type.
4. *notify flags*: The *notify flags* field contains a hex number of the form "0xNN" to be used in population the *fsinfo* data structure table entry for this file system type.
5. *function bitstring*: The *function bitstring* is a string of 28 0's and 1's. Each file system type potentially defines 28 functions to populate the *fstypsw* data structure table entry for itself. All file system types do not supply all the functions in this table, however, and this bitstring is used to indicate which of the functions are present and which are absent. A '1' in this string indicates that a function has been supplied, and a '0' indicates that a function has not been supplied. Successive characters in the string represent successive elements of the *fstypsw* data structure, with the first entry in this data structure represented by the rightmost character in the string.

SEE ALSO

sfsys(4).
idinstall(1m), idbuild(1m) in the *User's/System Administrator's Reference Manual.*

NAME

mtune – file format.

SYNOPSIS

mtune

DESCRIPTION

The *mtune* file contains information about all the system tunable parameters. Each tunable parameter is specified by a single line in the file, and each line contains the following whitespace-separated set of fields:

1. *parameter name*: A character string no more than 20 characters long. It is used to construct the preprocessor "#define's" that pass the value to the system when it is built.
2. *default value*: This is the default value of the tunable parameter. If the value is not specified in the *stune* file, this value will be used when the system is built.
3. *minimum value*: This is the minimum allowable value for the tunable parameter. If the parameter is set in the *stune* file, the configuration tools will verify that the new value is equal to or greater than this value.
4. *maximum value*: This is the maximum allowable value for the tunable parameter. If the parameter is set in the *stune* file, the configuration tools will check that the new value is equal to or less than this value.

The file *mtune* normally resides in */etc/conf/cf.d*. However, a user or an add-on package should never directly edit the *mtune* file to change the setting of a system tunable parameter. Instead the *idtune* command should be used to modify or append the tunable parameter to the *stune* file.

In order for the new values to become effective the UNIX System kernel must be rebuilt and the system must then be rebooted.

SEE ALSO

stune(4).

idbuild(1m), idtune(1m) in the *User's/System Administrator's Reference Manual*.

NAME

sdevice – file format.

SYNOPSIS

sdevice

DESCRIPTION

The *sdevice* file contains local system configuration information for each of the devices specified in the *mdevice* file. It contains one or more entries for each device specified in *mdevice*. *Sdevice* is present in the directory */etc/conf/cf.d*, and is coalesced from component files in the directory */etc/conf/sdevice.d*. Files in */etc/conf/sdevice.d* are the *System* file components either delivered with the base system or installed later via *idinstall*.

Each entry must contain the following whitespace-separated fields:

1. *Device name:* This field contains the internal name of the driver. This must match one of the names in the first field of an *mdevice* file entry.
2. *Configure:* This field must contain the character 'Y' indicating that the device is to be installed in the Kernel. For testing purposes, an 'N' may be entered indicating that the device will not be installed.
3. *Unit:* This field can be encoded with a device dependent numeric value. It is usually used to represent the number of subdevices on a controller or psuedo-device. Its value must be within the minimum and maximum values specified in fields 7 and 8 of the *mdevice* entry.
4. *Ipl*: The *ipl* field specifies the system ipl level at which the driver's interrupt handler will run in the new system kernel. Legal values are 0 through 8. If the driver doesn't have an interrupt handling routine, put a 0 in this field.
5. *Type:* This field indicates the type of interrupt scheme required by the device. The permissible values are:

   0 – The device does not require an interrupt line.

   1 – The device requires an interrupt line.
   If the driver supports more than one hardware controller, each controller requires a separate interrupt.

   2 – The device requires an interrupt line.
   If the driver supports more than one hardware controller, each controller will share the same interrupt.

   3 – The device requires an interrupt line.
   If the driver supports more than one hardware controller, each controller will share the same interrupt. Multiple device drivers having the same ipl level can share this interrupt.

6. *Vector:* This field contains the interrupt vector number used by the device. If the *Type* field contains a 0 (i.e. no interrupt required), this field should be encoded with a 0. Note that more than one device can share an interrupt number.

7. *SIOA:* The *SIOA* field (Start I/O Address) contains the starting address on the I/O bus through which the device communicates. This field must be within 0x1 and 0x3fff. (If this field is not used, it should be encoded with the value zero.)
8. *EIOA:* The field (End I/O Address) contains the end address on the I/O bus through which the device communicates. This field must be within 0x1 and 0x3fff. (If this field is not used, it should be encoded with the value zero.)
9. *SCMA:* The *SCMA* field (Start Controller Memory Address) is used by controllers that have internal memory. It specifies the starting address of this memory. This field must be within 0xa0000 and 0xfbfff. (If this field is not used, it should be encoded with the value zero.)
10. *ECMA:* The *ECMA* (End Controller Memory Address) specifies the end of the internal memory for the device. This field must be within 0xa0000 and 0xfbfff. (If this field is not used, it should be encoded with the value zero.)

SEE ALSO

mdevice(4).
idinstall(1m) in the *User's/System Administrator's Reference Manual.*

NAME

sfsys – file format.

SYNOPSIS

sfsys

DESCRIPTION

The *sfsys* file contains local system information about each file system type specified in the *mfsys* file. It is present in the directory */etc/conf/cf.d*, and contains a one-line entry for each file system type specified in the *mfsys* file. The *sfsys* file is coalesced from component files in the directory */etc/conf/sfsys.d*. Each line in this file is a whitespace-separate set of fields that specify:

1. *name*: This field contains the internal name of the file system type (e.g., DUFST, S51K). By convention, this name is up to 32 characters long, and is composed of all uppercase alphanumeric characters.
2. *Y/N*: This field contains either an uppercase 'Y' (for "yes") or an uppercase 'N' (for "no) to indicate whether the named file system type is to be configured into the next system kernel to be built.

SEE ALSO

mfsys(4).
idinstall(1m), idbuild(1m) in the *User's/System Administrator's Reference Manual*.

NAME

stune – file format.

SYNOPSIS

stune

DESCRIPTION

The *stune* file contains local system settings for tunable parameters. The parameter settings in this file replace the default values specified in the *mtune* file, if the new values are within the legal range for the parameter specified in *mtune*. The file contains one line for each parameter to be reset. Each line contains two whitespace-separated fields:

1. *external name*: This is the external name of the tunable parameter used in the *mtune* file.
2. *value*: This field contains the new value for the tunable parameter.

The file *stune* normally resides in */etc/conf/cf.d*. However, a user or an add-on package should never directly edit the *mtune* file. Instead the *idtune* command should be used.

In order for the new values to become effective the UNIX kernel must be rebuilt and the system must then be rebooted.

SEE ALSO

mtune(4).
idbuild(1m), idtune(1m) in the *User's/System Administrator's Reference Manual*.

B Appendix B

A Simple Game Port Driver

Game Port Driver Example

The Game Port or Game Controller allows one or more joystick or other positional indicating devices to be plugged into a PC. Game controller boards are available from several manufacturers, and a large base of MS-DOS software (mostly video games) already exists. Since the Game Controller hardware is so simple, it is useful as the basis for an elementary UNIX device driver. A UNIX driver would allow a UNIX process to be driven by joystick position inputs.

Although the I/O Address Map assigns Hex address 200 through 20f to Game Control, the Games Controller peripheral board uses only address 201. The interface is initialized by a write to address 201 and provides position inputs on successive reads of address 201. The position readings indicate the position of the joystick as follows:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Joy A But 1	Joy A But 2	Joy B But 1	Joy B But 2	Joy A Y Pos	Joy A X Pos	Joy B Y Pos	Joy B X Pos

Where `Joy A` and `Joy B` represent the two joystick inputs, and `But 1` and `But 2` represent the mark and fire buttons.

With such an interface, the driver needs no interrupt entry point (IVN 2 is, however, reserved for the Games Controller and is often referred to as the "Games Interrupt"). The following UNIX driver is implemented without assigning any interrupt vector to the Controller and simply polls the interface upon a read() call from the Operating System.

The main points to extract from review of this driver are an understanding of basic driver structure and the use of printf and trace driver calls used for driver debugging.

- Lines 29-35:
  Show a way to assign values to ioctl() *cmd* fields. By convention, the left shifting of the first letter of the device name and or-ing low-order values helps to randomize ioctl() arguments across device drivers and provides some protection if a user process or program opens the wrong special file. Since the ioctl values must be known by user processes to set and clear debugging or tracing, it would be advisable to put them in a header file (for example, **/usr/include/sys/game.h**) so that both the driver and user programs can access them.

- Line 53:
  Shows using the minor() macro in **/usr/include/sys/sysmacros.h** to strip off the upper bits of *dev* to determine the minor device specified by the open() system call.

- Line 54:
  Puts a printf on **/dev/console** when **/dev/game** is opened if the GAMEDBSET ioctl() call has been made.

- Line 55:
  Makes a call to trsave() (line 161), which calls the trace driver (see Appendix C) if the GAMETRSET ioctl() call has been made.

- Lines 82-114
  The read() routine. Note the use of the device prefix in the entry point, *gameread()* and also the use of outb(), inb(), spl(), and copyout(), as described earlier.

- Lines 154-159:
  Implement the conditional use of printf's for debugging.

- Lines 161-189:
  Implement the conditional use of the trace driver for debugging. See Appendix C.

```
1       /*      Copyright (c) 1985 AT&T */
2       /*        All Rights Reserved   */
3
4       /*      A Simple Game Port Driver
5        *
6        * The Game Port or Game Controller allows one or more joystick or
7        * other positional indicating devices to be plugged into a PC.
8        *
9        * Although the I/O Address Map assigns Hex address 200 through 20f to
10       * Game Control,
11       * the Games Controller peripheral board uses only address 201. The
12       * interface is initialized by a write to address 201 and provides
13       * position inputs on successive reads of address 201. The position
14       * readings indicate the position of the joystick as follows:
15       *
16       * This driver contains calls to the trace driver (See Appendix C.).
17       *
18      */
19
20      #include "sys/types.h"
21      #include "sys/param.h"
22      #include "sys/dir.h"
23      #include "sys/signal.h"
24      #include "sys/errno.h"
25      #include "sys/ioctl.h"
26      #include "sys/user.h"
27      #include "sys/sysmacros.h"
28
29      /* ioctl commands */
30      #define GAME    ('g'<<8)
31      #define GAMEDBSET       (GAME|01)
32      #define GAMEDBCLR       (GAME|02)
33      #define GAMEDELAY       (GAME|03)
34      #define GAMETRSET       (GAME|04)
35      #define GAMETRCLR       (GAME|05)
36
37      /* flag values */
38      #define OPEN    01
39      #define DBG     02
40      #define TRACE   04
41
42      char space[10];
43      struct game {
```

```
44              int    flag;
45              int delay;
46      } game_dev;
47
48      gameopen(dev, mode)
49      int dev, mode;
50      {
51              int port;
52
53              port=minor(dev);
54              gamedbprt("gamedev: in open");
55              gamesave(port,'O',0 ,0);
56              if (port != 0) {
57                      u.u_error = ENXIO;
58                      return;
59              }
60
61              if (!(game_dev.flag&OPEN)) {
62                      game_dev.flag |= OPEN;
63                      game_dev.delay =2;
64              }
65      }
66
67      gameclose(dev)
68      int dev;
69      {
70              int port;
71
72              port=minor(dev);
73              gamedbprt("gamedev: in close");
74              gamesave(port,'C',0 ,0);
75              if (port != 0) {
76                      u.u_error = ENXIO;
77                      return;
78              }
79              game_dev.flag &= ~OPEN;
80      }
81
82      gameread(dev)
83      {
84              int ii, x, del, junk;
85              int port;
86
87              port=minor(dev);
88              gamesave(port,'R',u.u_count,0);
```

```
89              if (port != 0) {
90                      u.u_error = ENXIO;
91                      return;
92              }
93
94              /* Write to the Game Controller to trigger position circuits,
95                 then read the Game Port Interface (with a slight delay
96                 between read cycles) to report position of joystick
97                 or other game control device.                */
98
99              outb (0x201, 0);
100             for (ii=0; ii <= 9; ii++){
101                     for (del=1; del<=game_dev.delay; del++)
102                             junk++;
103                     space[ii]=inb(0x201);
104             }
105             if (u.u_count) {
106                     x=spl4();
107                     if (copyout ( space, u.u_base, u.u_count) ){
108                             gamedbprt ("/dev/game: read error");
109                             u.u_error = ENXIO;
110                     }
111                     splx(x);
112                     u.u_count = 0;
113             }
114     }
115
116     gamewrite(dev)
117     {
118             int port;
119
120             port=minor(dev);
121             gamedbprt ("/dev/game: entered write routine");
122             gamesave(port,'W',u.u_count,0);
123     }
124
125     gameioctl(dev, cmd, arg )
126     int     dev;
127     unsigned int    cmd;
128     unsigned int    arg;
129     {
130             int port;
131
132             port=minor(dev);
133             gamesave(port,'I',cmd, arg);
```

```
134             switch(cmd) {
135                     case GAMEDBSET:
136                             game_dev.flag |= DBG;
137                             break;
138                     case GAMEDBCLR:
139                             game_dev.flag &= ~DBG;
140                             break;
141                     case GAMEDELAY:
142                             game_dev.delay = arg;
143                             break;
144                     case GAMETRSET:
145                             game_dev.flag |= TRACE;
146                             break;
147                     case GAMETRCLR:
148                             game_dev.flag &= ~TRACE;
149                             break;
150                     default:
151                             u.u_error = EINVAL;
152             }
153     }
154     gamedbprt(str)
155     char *str;
156     {
157             if (game_dev.flag&DBG)
158                     printf("%s\n", str);
159     }
160
161     gamesave(port, type, word1, word2)
162     int     port, word1, word2;
163     unsigned char   type;
164     {
165             static int      gameseqn;
166             register sps;
167             struct {
168                     short   e_seqn;
169                     char    e_type;
170                     char    e_dev;
171                     short   e_word1;
172                     short   e_word2;
173             } gameent;
174
175             if (!(game_dev.flag&TRACE))
176                     return;
177
178
```

```
179             sps = spl5();
180             if (gameseqn >= 077777)
181                     gameseqn = 0;
182             gameent.e_seqn = ++gameseqn;
183             gameent.e_type = type;
184             gameent.e_dev = port;
185             gameent.e_word1 = word1;
186             gameent.e_word2 = word2;
187             trsave(0, port%16, (char *) & gameent, sizeof(gameent));
188             splx(sps);
189     }
```

C Appendix C

The Trace Driver

The Game Port Driver in Appendix B introduced the basic structure of device drivers, but now let's look at something a bit more practical. The trace driver is a pseudo-device that allows the 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(). An example of how another driver does this is given in Appendix B. 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 the 386 System) 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 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 NIL0377
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 TRACRCD:
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_ct--)
144                             passc(getc(&tr_p->tr_outq));
145                     tr_p->tr_chno = NIL;
146                     tr_p->tr_rcnt--;
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;
```

```
161             tr_p = &tr_data[dev];
162             ct &= 0377;
163             if ((tr_p->tr_chbits&(1<<chno)) == 0)
164                     return;
165             if ((tr_p->tr_outq.c_cc + ct + 2 + sizeof(lbolt)) >TRQMAX)
166                     return;
167             putc(chno, &tr_p->tr_outq);
168             putc(ct + sizeof(lbolt), &tr_p->tr_outq);
169             cpt = (char *)&lbolt;
170             for (n = 0; n < sizeof(lbolt); ++n)
171                     putc(*cpt++, &tr_p->tr_outq);
172             for (n=0;n<ct;n++)
173                     putc(buf[n], &tr_p->tr_outq);
174             tr_p->tr_rcnt++;
175             if (tr_p->tr_state&TRSLEEP) {
176                     tr_p->tr_state &= ~TRSLEEP;
177                     wakeup((caddr_t)&tr_p->tr_rcnt);
178             }
179     }
```

```
180  /*  Copyright (c) 1985 AT&T                                          */
181  /*    All Rights Reserved                                            */
182  /*                                                                   */
183  /*  trsav - save 386unix event traces                                */
184  /*                                                                   */
185  /*   usage: trsav mask device                                        */
186  /*                                                                   */
187  /*   Trsav opens the minor device of the trace driver specified      */
188  /*   by "device," enables the channels specified by "mask" (octal),*/
189  /*   and then reads event records and writes them to its standard    */
190  /*   output (unformatted) until killed.  Bit 0 of mask enables       */
191  /*   channel zero, bit 1 channel one, etc.                           */
192  /*                                                                   */
193  /*   For example, to enable saving of trace channel 0 from minor     */
194  /*   device 0 of the trace driver and save the output in a file in */
195  /*   /tmp, the following command can be used:                        */
196  /*                                                                   */
197            /*trsav 1 /dev/trace0 > /tmp/temp.file &                   */
198
199  #include <stdio.h>
200  #include "sys/types.h"
201  #include "sys/tty.h"
202  #include "sys/trace.h"
203
204  char ev[512];
205  main(argc, argv)
206  char *argv[ ];
207  {
208            int fd, n, k, seqno, chbits;
209            if (argc != 3) {
210                     fprintf(stderr, "Incorrect number of arguments\n");
211                     fprintf(stderr, "Usage: trsav mask device\n");
212                     exit(1);
213            }
214            if ((fd = open(argv[2], 2)) < 0) {
215                     perror("trsav open:");
216                     exit(2);
217            }
218            setbuf(stdout, NULL);
219            sscanf(argv[1], "%6o", &chbits);
220            if ((k = ioctl(fd, TRASETC, chbits)) < 0) {
221                     perror("trsav ioctl:");
222                     exit(3);
223            }
```

```
224             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     }
```

```
/*  Copyright (c) 1985 AT&T                                          */
/*    All Rights Reserved                                            */
/*                                                                   */
/*  386unix trace.h driver header file.                              */
/*                                                                   */
/*  When the IOCTL defines are provided in a .h file,                */
/*  both the driver and any user programs that need the IOCTL        */
/*  values can work from the same set of #defines.                   */
/*                                                                   */
/*                                                                   */

/* IOCTL defines */
#define TRAC                ('T'<<8)
#define TRACMD              (TRAC|8)
#define TRAERRS             (TRAC|9)
#define TRARPT              (TRAC|10)
#define TRASDEV             (TRAC|11)
#define TRAATTACH           (TRAC|11)
#define TRADETACH           (TRAC|31)
#define TRAERRSET           (TRAC|32)
#define TRAERRGET           (TRAC|34)
#define TRAOPTS             (TRAC|33)
#define TRAPCIDPTS          (TRAC|35)
#define TRACRCD             (TRAC|16)
#define TRAGETC             (TRAC|17)
#define TRASETC             (TRAC|18)
#define TRACLRC             (TRAC|19)
#define TRASTAT             (TRAC|36)

/*
* Per trace structure
*/
struct trace {
            struct          clist tr_outq;
            short           tr_state;
            short           tr_chbits;
            short           tr_rcnt;
            unsigned char   tr_chno;
            char            tr_ct;
};
```

```
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                              */
289  /*                or                                                  */
290  /*                trsav mask device | trfmt                           */
291  /*                                                                    */
292  /*   This version will format and print predefined lines of text      */
293  /*   for only a few types of typical driver traces: O is "open,"      */
294  /*   C is "close," etc. If you wish to use other trace points in      */
295  /*   your driver, define your own trace identifiers and add them      */
296  /*   to the case statement below.                                     */
297  /*                                                                    */
298  /*                                                                    */
299
300  #include <stdio.h>
301
302  #define    MASK16          0177777
303  #define    SSTOL(x, y)     ((((long)x)<<16)|(((long)y)&MASK16))
304
305  struct event {
306            unsigned short  lbolt1;
307            unsigned short  lbolt2;
308            unsigned short  seq;
309            unsigned char   typ;
310            unsigned char   dev;
311            unsigned short  wd1;
312            unsigned short  wd2;
313  } ev;
314  main(argc, argv)
315  char *argv[ ];
316  {
317            extern int optind;
318            int x, fd, k, n, seqno, con;
319            char *type;
320            long time1;
```

```
321             char xxx;
322
323     setbuf(stdout,NULL);
324             seqno = 1;
325             for (;;) {
326                     x = getchar();
327                     n = getchar();
328                     if ((k=fread((char *)&ev, sizeof(xxx), n, stdin))< 0){
329                             perror();
330                             exit(3);
331                     }
332                     if (k == 0) {
333                             clearerr(stdin);
334                             sleep(1);
335                             continue;
336                     }
337                     if (ev.seq != seqno)
338                             printf("**%d event records lost**0,
339                                     ev.seq - seqno);
340                     seqno = ev.seq + 1;
341                     if (k == 12) {
342                             time1 = STOL(ev.lbolt2, ev.lbolt1);
343                             printf(" %10lu%6d", time1, ev.seq);
344                             switch((int)ev.typ){
345                             case 'W':
346                                     type = "write";
347                                     printf(" %-8s %2o%6o%6o", type, ev.dev,
348                                     ev.wd1, ev.wd2);
349                                     break;
350                             case 'R':
351                                     type = "read";
352                                     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%6u%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 }
```

D Appendix D

A Prototype Floppy Disk Driver

The attached source code presents some selected portions of a prototype PC floppy disk device driver. This is not the entire source file, and some aspects of this driver are not representative of more general drivers. For example, since the floppy driver contains many data structures that are similar or identical to a companion hard disk driver, some common data structures are shared by the two device drivers, and the block major device number is used to access the floppy and hard disk portions of those data structures. Additionally, some of the function calls made inside the floppy driver are to functions that have been deleted for brevity or are functions defined in other source files.

Despite these restrictions, the floppy device driver is a good example of a rather complex driver. This driver also shows how a driver can implement both block and character (raw) device I/O. As a block device driver, the UNIX File System accesses the device through the driver strategy routine (see PCf_strategy() on line 211). Since the floppy driver also acts as a character device, the "raw" I/O driver entry points (PCf_read() on line 369 and PCf_write() on line 390) are also provided. You can see that the PCf_read() and PCf_write() routines make use of physio(), which, in turn, calls the floppy strategy routine.

```
1   /*
2    *      Copyright (c) 1987 AT&T
3    *  All Rights Reserved
4    *
5    *      These procedures define portions of a Prototype PC floppy disk driver:
6    *
7    *      NOTE: THIS IS NOT THE COMPLETE DRIVER SOURCE CODE; ONLY REPRESENTATIVE
8    *      SECTIONS ARE INCLUDED AS EXAMPLES.
9    *
10   *              PCf_open:  Opens a unit by setting flags, initializing variables,
                            and initializing structures
11   *              PCf_close: Closes a unit by resetting flags or possibly flushing
                            buffers or queues
12   *              PCf_strategy: Validates requests, queues it on device queue,
                            tries to start I/O
13   *              PCf_intr:  Processes interrupts due to access completion, seek end,
                            or spurious causes
14   *              PCf_read:  Performs raw read (uses physio routine)
15   *              PCf_write: Performs raw write (uses physio routine)
16   *              PCf_ioctl: Special functions - format, etc.
17   *
```

```
18          *       Internal routines:
19          *               PCf_init:  Initializes device at boot time
20          *
21          *               (Other internal routines have been deleted for brevity)
22          */
23
24      #include <sys/signal.h>
25      #include <sys/types.h>
26      #include <sys/sysmacros.h>
27      #include <sys/param.h>
28      #include <sys/systm.h>
29      #include <sys/buf.h>
30      #include <sys/iobuf.h>
31      #include <sys/conf.h>
32      #include <sys/dir.h>
33      #include <sys/user.h>
34      #include <sys/errno.h>
35      #include <sys/elog.h>
36      #include <sys/open.h>
37      #include <sys/file.h>
38      #include "sys/PCf_disk.h"
39
40      #define FMAJ 1                  /* MAJOR DEVICE-used in minor device macros */
41                                      /* Floppy assigned Block Major=1; needed since  */
42                                      /* floppy and wini(hard disk) share data structures */
43      char S5fraw_buf[512];
44      long Fmt_sec;
45      long            physaddr();
46      extern struct PCd_dev       PCf_dev[];    /* device-data-structures */
47      extern struct iobuf         PCf_tab[];    /* buffer header */
48      extern struct PCd_dev *PCf_i_tab[];       /* intr -> device mapping */
49      extern struct buf           PCf_rbuf[];   /* raw buffer header */
50      extern struct PCd_cdrt PCdf[F_NDRTAB];    /* drtabs */
51      extern struct fparm_tab ibmf512[];        /* floppy disk parameter table */
52      extern struct PCd_minor PCd_major[][F_MAXMINOR];/* major/minor number bit map */
53      extern int    fcntr_state;                /* current state of controller*/
54      extern int    activefloppy;               /* active floppy I/O */
55      extern        long          ftransvector; /* base address of memory transfer area */
56      extern        int           ftrans_cnt;   /* # of sectors being transferred */
57      extern        int           fsec_cnt;     /* total # sectors transferred so */
                                                  /*far for a buf structure */
58      struct iotime PCfstat[F_NPER_CONTR];
59      extern char Flastdev[F_NPER_CONTR];
60      extern char Ftryflop[F_NPER_CONTR];
61
62      PCf_tmout()
63      {
64              /* Stub for timeout routine */
65      }
```

```
66
67      /*
68       * PCf_init
69       *       Call at boot time to init device.
70       *       This routine sets up the mmu and intr tables for the ROM based
71       *       disk routines.  Recalibration will be done by boot strap loader.
72       */
73      PCf_init()
74      {
75              int i;
76              struct iobuf *iobuf;
77              char arg;
78
79              /* Initialize data structure constants */
80              for(i=0; i < F_NPER_CONTR; i++) {
81              PCf_dev[i].d_state.s_bufh = &PCf_tab[i];/* buffer header */
82              PCf_dev[i].d_state.s_devcode = i; /* device code */
83              PCf_dev[i].d_state.s_level = F_INTLVL;/* local copy */
84              PCf_dev[i].d_state.s_active = IO_IDLE;/*active flag */
85              fcntr_state = IO_IDLE;
86              iobuf = &PCf_tab[i];
87              iobuf->b_forw = (struct buf *)iobuf;/* initialize buffer */
88              iobuf->b_back = (struct buf *)iobuf;
89              iobuf->b_actf = (struct buf *)iobuf;/* headers as in 310 */
90              iobuf->b_actl = (struct buf *)iobuf;
91              Flastdev[i] = F96DEV;
92              Ftryflop[0] = 0;
93              Ftryflop[1] = 0;
94          }
95          timeout(PCf_tmout,&arg,3*HZ);
96      }
97
98      /*
99       * PCf_open
100      *       Open a unit.
101      *
102      * Sets up given partition as open.
103      */
104     PCf_open(dev, flag, otyp)
105     dev_t   dev;
106     int     flag;
107     unsigned otyp;
108     {
109             register struct PCd_dev *dd;
110             register struct iobuf*bufh;
111             register struct buf *bp;
112             register unsigned board;
113             unsigned unit, x;
114             static int firsttime = 1;
```

```
115             char *dp;
116             int i, fret;
117
118             if(minor(dev) > F_MAXMINOR) {
119                     u.u_error = ENXIO;
120                     return;
121             }
122             /*
123              * if this is the first open ever then initialize
124             */
125             if (firsttime) {
126                     firsttime--;
127                     PCf_init();
128             }
129             board = UNIT(FMAJ,dev);
130             dd = &PCf_dev[board];
131             bufh = dd->d_state.s_bufh;
132             if ((dd->d_state.s_flags & SF_OPEN) == 0) {
133                     bp = &PCf_rbuf[UNIT(FMAJ,dev)];
134                     x = spl5();
135                     while(bp->b_flags&B_BUSY) {
136                         bp->b_flags |= B_WANTED;
137                         sleep((caddr_t)bp,PRIBIO);
138                     }
139                     bp->b_flags = B_BUSY | B_READ;
140                     splx(x);
141                     if ((dd->d_state.s_flags & SF_OPEN) == 0) {
142                         dd->d_state.s_flags = SF_OPEN | SF_READY | RESETTING;
143                         bufh->b_active = IO_BUSY;
144                         u.u_error = PCf_sweep(dd, dev, flag, bp);
145                         bp->b_flags &=  (B_BUSY | B_READ);
146                         if (bp->b_flags&B_WANTED)
147                             wakeup((caddr_t)bp);
148                         if (u.u_error == 0)
149                             dd->d_state.s_flags = SF_OPEN|SF_READY;
150                         else {
151                             dd->d_state.s_flags = 0;
152                             return;
153                         }
154                         for ( i=0; i<OTYPCNT; i++ )
155                             dd->d_state.s_popen[i] = 0;
156                         PCf_start(dd);
157                     }
158             }
159             dp = &dd->d_state.s_popen[0];
160             if ( otyp == OTYP_LYR )
161                     ++dp[OTYP_LYR];
162             else if ( otyp < OTYPCNT )
163                     dp[otyp] |= (1 << PARTITION(FMAJ,dev));
```

```
164     }
165
166     /*
167      * PCf_close
168      *       Close a unit.
169      *
170      * Called on last close of a partition; thus, "close" the partition.
171      * If this was last partition, make the unit closed & not-ready.
172      * In this case, next open will re-initialize.
173      */
174     PCf_close(dev, flag, otyp)
175     register dev_t                  dev;
176     int flag;
177     unsigned otyp;
178     {
179             register struct PCd_dev     *dd;
180             extern                      dev_t rootdev;
181             struct buf                  *bufh;
182             char *dp;
183             int i;
184
185             if (dev == rootdev)
186                     return;  /* never close rootdev */
187             dd = &PCf_dev[UNIT(FMAJ,dev)];
188             /*
189              * Close the partition. If last partition, close the unit.
190              */
191             dp = &dd->d_state.s_popen[0];
192             if ( otyp == OTYP_LYR )
193                     --dp[OTYP_LYR];
194             else if ( otyp < OTYPCNT )
195                     dp[otyp] &=  (1 << PARTITION(FMAJ,dev));
196             for ( i=0; i<OTYPCNT && dp[i]==0; i++ );
197             if ( i == OTYPCNT ) /* only close if closed for all types of open */
198                     dd->d_state.s_flags = 0;
199     }
200
201     /*
202      * PCf_strategy
203      *       Queue an I/O Request, and start it if not busy already.
204      *
205      * Check legality, and adjust for partitions.  Reject request if unit is not-ready.
206      *
207      * Note: The partition-check algorithm insists that requests must not cross
208      *        a sector boundary.  If partition size is not a multiple of BSIZE, the
209      *        last few sectors in the partition are not accessible.
210      */
211     PCf_strategy(bp)
212     register struct buf *bp;
```

```
213  {
214          register struct PCd_dev     *dd;
215          register struct PCd_drtab *dr;
216          register struct PCd_cdrt *cdr;
217          register struct PCd_part *p;
218          struct   iobuf     *bufh;
219          struct   buf       *ap;
220          daddr_t            secno;
221          unsigned           x;
222          char drive;
223
224          drive = UNIT(FMAJ,bp->b_dev);
225          dd = &PCf_dev[drive];
226          dr = &dd->d_drtab;
227          p = &dr->dr_part[PARTITION(FMAJ,bp->b_dev)];
228          PCfstat[drive].io_bcnt += btoc(bp->b_bcount);
229          bp->b_start = lbolt;
230          /*
231           * Figure "secno" from b_blkno. Check if ready, and see if fits in partition.
232           * Adjust sector # for partition.
233           *
234           * Note: if format, b_blkno is already the correct sector number.
235           */
236          secno = bp->b_blkno;
237          if ( Ftryflop[drive] ) {
238              if ( secno >= p->p_nsec ||
239                 (secno+(bp->b_bcount+dr->dr_secsiz-1)/dr->dr_secsiz) > p->p_nsec){
240                       if ( Flastdev[drive] == F96DEV ) {
241                                cdr = &PCdf[F48DEV];
242                                Flastdev[drive] = F48DEV;
243                       }
244                       else {
245                                cdr = &PCdf[F96DEV];
246
247                                Flastdev[drive] = F96DEV;
248                       }
249                       dr->dr_ncyl = cdr->cdr_ncyl;
250                       dr->dr_nhead = cdr->cdr_nhead;
251                       dr->dr_nsec = cdr->cdr_nsec;
252                       dr->dr_spc = dr->dr_nhead * dr->dr_nsec;
253                       dr->dr_secsiz = cdr->cdr_secsiz;
254                       dr->dr_part = cdr->cdr_part;
255                       p = &dr->dr_part[PARTITION(FMAJ,bp->b_dev)];
256              }
257          }
258          if (((dd->d_state.s_flags & SF_READY) == 0)
259              || (secno > p->p_nsec)) {
260                   /* not ready or off end */
261                   bp->b_flags |= B_ERROR;
```

```
262                     bp->b_error = ENXIO;                    /* bad block */
263                     x = spl5();
264                     iodone(bp);                             /* return buffer */
265                     splx(x);
266                     return;
267             }
268             if (secno == p->p_nsec) {
269                     if (bp->b_flags & B_READ)
270                             bp->b_resid = bp->b_bcount;
271                     else {
272                             bp->b_error = ENXIO;
273                             bp->b_flags |= B_ERROR;
274                     }
275                     x = spl5();
276                     iodone(bp);
277                     splx(x);
278                     return;
279             }
280             if ((secno + (bp->b_bcount+dr->dr_secsiz-1)/dr->dr_secsiz) > p->p_nsec) {
281                     /* just asked to read last one. Send EOF */
282                     bp->b_resid = bp->b_bcount;
283                     x = spl5();
284                     iodone(bp);
285                     splx(x);
286                     return;
287             }
288             secno += p->p_fsec;
289             bp->b_resid = p->p_fsec/dr->dr_spc;/* starting cylinder of slice */
290             /*
291              * Add request to queue, & (maybe) start it.
292              */
293             x = spl5();
294             bufh = dd->d_state.s_bufh;
295             ap = bufh->b_forw;
296             /*
297              * find the right place to put this buffer into the list by cylinder number
298              */
299             while( ap != bufh->b_back) {
300     if( (bp->b_blkno+(bp->b_resid*dr->dr_spc)) < (ap->b_blkno+(ap->b_resid*dr->dr_spc)) )
301                             break;
302                     else
303                             ap = ap->av_forw;
304             }
305             if ( ap == (struct buf *)bufh ) {
306                     /* no list currently exists - start one */
307                     bufh->b_actf = bp;
308                     bufh->b_forw = bp;
309                     bufh->b_back = bp;
310                     bp->av_forw = bp;
```

```
311                     bp->av_back = bp;
312             }
313             else if ( ap == bufh->b_back ) {
314     if ( (bp->b_blkno+(bp->b_resid*dr->dr_spc)) < (ap->b_blkno+(ap->b_resid*dr->dr_spc)) ) {
315                             bp->av_back = ap->av_back;
316                             ap->av_back->av_forw = bp;
317                             bp->av_forw = ap;
318                             ap->av_back = bp;
319                             if ( bufh->b_forw == ap )
320                                     bufh->b_forw = bp;
321                     }
322                     else {
323                             ap->av_forw->av_back = bp;
324                             bp->av_forw = ap->av_forw;
325                             ap->av_forw = bp;
326                             bp->av_back = ap;
327                             bufh->b_back = bp;
328                     }
329             }
330             else {
331                     bp->av_back = ap->av_back;
332                     ap->av_back->av_forw = bp;
333                     bp->av_forw = ap;
334                     ap->av_back = bp;
335                     if ( bufh->b_forw == ap )
336                     bufh->b_forw = bp;
337             }
338             if (fcntr_state == IO_IDLE)
339                     PCf_start(dd,x);
340             splx(x);
341     }
342
343     PCf_intr(dev)
344     int     dev;
345     {
346             extern char seek_status;
347             register struct PCd_dev *dd;
348             struct iobuf  *bufh;
349             unsigned int x;
350             static int save_state=0;
351
352             seek_status |= 0x80;
353             dd = &PCf_dev[activefloppy];
354             bufh = dd->d_state.s_bufh;/* get buf-header, too */
355             if(dd->d_state.s_active == IO_BUSY)
356                     /* call BIOS Hardware Interrupt service routine */
357                     flpy_hwintr();
358             else
359                     save_state=dd->d_state.s_active;
```

```
360     }
361
362     /*
363      * PCf_read
364      *       "Raw" read.  Use physio().
365      *
366      * Calls:
367      *       PCf_strategy (indirectly, thru physio)
368      */
369     PCf_read(dev)
370     dev_t             dev;
371     {
372             register struct PCd_dev *dd;
373             register struct PCd_drtab *dr;
374             register struct PCd_part *p;
375
376             dd = &PCf_dev[UNIT(FMAJ,dev)];
377             dr = &dd->d_drtab;
378             p = &dr->dr_part[PARTITION(FMAJ,dev)];
379             if ( physck(p->p_nsec,B_READ) )
380                     physio(PCf_strategy, &PCf_rbuf[UNIT(FMAJ,dev)], dev, B_READ);
381     }
382
383     /*
384      * PCf_write
385      *       "Raw" write.  Use physio().
386      *
387      * Calls:
388      *       PCf_strategy (indirectly, thru physio)
389      */
390     PCf_write(dev)
391     dev_tdev;
392     {
393             register struct PCd_dev *dd;
394             register struct PCd_drtab *dr;
395             register struct PCd_part *p;
396
397             dd = &PCf_dev[UNIT(FMAJ,dev)];
398             dr = &dd->d_drtab;
399             p = &dr->dr_part[PARTITION(FMAJ,dev)];
400             if ( physck(p->p_nsec,B_WRITE) )
401                     physio(PCf_strategy, &PCf_rbuf[UNIT(FMAJ,dev)], dev, B_WRITE);
402     }
403
404     /*
405      * PCf_ioctl
406      */
407     PCf_ioctl(dev, cmd, cmdarg, flag)
408     dev_t             dev;
```

```
409     int             cmd;
410     char            *cmdarg;
411     int             flag;
412     {
413             register struct buf *bp;
414             register struct PCd_dev *dd;
415             register struct PCd_drtab *dr;
416             register struct PCd_part *p;
417             unsigned x;
418             char j, k, *tblptr;
419             int *cmdint;
420
421             dd = &PCf_dev[UNIT(FMAJ,dev)];
422             dr = &dd->d_drtab;
423             bp = &PCf_rbuf[UNIT(FMAJ,dev)];
424             p = &dr->dr_part[PARTITION(FMAJ,dev)];
425
426             switch (cmd) {
427             case FMTFLPY:
428                     /* ....
429                        ....  Specific ioctl code deleted
430                        ....
431                        break;
432             case ....
433                        ....
434                        ....  Specific ioctl code deleted
435                        ....
436                        break;*/
437             default:
438                     u.u_error = ENXIO;
439                     return;
440             }
441     }
442
```

```
443     /*      Copyright (c) 1984 AT&T                                 */
444     /*        All Rights Reserved                                   */
445
446     /*      THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T     */
447     /*      The copyright notice above does not evidence any        */
448     /*      actual or intended publication of such source code.     */
449     /*@(#)1.3.1.6*/
450
451     /*
452      * disk.h
453      */
454     #include "sys/open.h"
455
456     /*********************************************/
457     /********    MCS ADDED FOR VTOC STUFF*******/
458     #define MAXBAD99
459     #define UNIXOs99/* system indicator for UNIX partition */
460     #defineALTMGK10x55
461     #define ALTMGK20xAA
462
463     /*** DEFINES TO SUPPORT THE VARIABLE SIZE OF ALTERNATE TRACKS ***/
464     #define MAXUPTO40  64      /* Max # of bad tracks for disks of 40 MB or less   */
465     #define MAXOVER40  99      /* Max # of bad tracks for disks greater than 40 MB */
466     #define NSECINA40  85000   /* Number of sectors in a 40 MB disk */
467                                /* 83385 sectors for a 40MB with 981 cyl & 5 heads  */
468                                /* 83640 sectors for a 40MB with 820 cyl & 6 heads  */
469     /** FLAGS FOR PCw_io ROUTINES **************/
470
471     #define B_FMTBAD           020000          /* must NOT overlap see buf.h! */
472     #define B_FMTTRK           030000          /* must NOT overlap see buf.h! */
473     #define B_RECOVR           060000          /* as above - used for recovery io */
474     #define B_FMTMSK           070000          /* mask for above */
475
476     /*** DEFINES FOR CASE STATEMENTS IN IOCTL ROUTINES ************/
477
478     #define RDDPARM            0
479     #define FMTBAD             1
480     #define FMTVERIFY          2
481     #define FMTFLPY            3
482     #define RDPARTBL           4
483     #define WRPARTBL           5
484     #define WRALTBL            6
485     #define WRBOOT             7
486     #define DOFMT              8
487     #define DOVRFY             9
488     #define W_RECOVER          10
489     #define RDALTBL            11
490     #define FMTEND             -1
```

```
491
492     /*** DEFINES FOR WINI VTOC ********************/
493
494     #define VSANITY             0xAA55
495     #define VVERSION            1
496     #define WSECSIZ             512
497     #define SEC_TRK             17
498     #define VNOWRITE            0
499     #define VNORM               1
500
501     /*****************************************/
502     #define FDUALDEV             2      /* drtab of either 96 or 48tpi floppy */
503     #define F48DEV               4      /* drtab for 48tpi, 9 sec/trk     */
504     #define F96DEV               3      /* drtab for 96tpi, 15 sec/trk    */
505
506     #define F_NDRTAB            5
507     #define F_NPART             2
508     #define F_NPER_CONTR        2       /* number of drivers per controller */
509     #define F_MAXMINOR          F_NDRTAB*F_NPART*F_NPER_CONTR
510                                         /* maximum minor # for floppy driver */
511     #define W_NDRTAB            1
512     #define W_NPART             5
513     #define W_NPER_CONTR        2       /* number of drivers per controller */
514     #define W_MAXMINOR          F_MAXMINOR/* maximum minor # for wini driver */
515                                         /* assumes more or equal floppy devs.*/
516     #define FBADSPEED           0x0200      /* wrong floppy speed error return */
517     #define FCMDERROR           0x0100      /* wrong floppy type error return */
518     #define FWRPROT             0x0300      /* write protect floppy error return */
519     #define WBADTRK             0x0B00      /* Wini bad track */
520     #define WUNERR              0x1000      /* Wini unrecoverable error */
521     #define WADRMRK             0x0200      /* Wini address mark not found */
522     #define WECCERR             0x1100      /* corrected ecc error */
523     #define WSEEKERR            0x4000      /* seek error */
524     #define DOS_SLICE           4           /* Major Minor of E drive */
525     #define S5WRETRY             5          /* wini retry count */
526     #define S5FRETRY             25         /* floppy retry count */
527     #define F_INTLVL             6          /* floppy interrupt level */
528     #define W_INTLVL             5          /* wini   interrupt level */
529
530     structPCd_minor {
531             unsigned partition: 4;      /* partition number */
532             unsigned drtab:4;           /* alternate drtab's */
533             unsigned unit: 4;           /* unit number */
534     };
535
536     #define  UNIT(maj,dev)      (PCd_major[maj][minor(dev)].unit)
537                                         /* dev -> unit# map I003 */
538     #define  DRTAB(maj,dev)     (PCd_major[maj][minor(dev)].drtab)
539                                         /* dev -> drtab-index map I003 */
```

```
540     #define  PARTITION(maj,dev)     (PCd_major[maj][minor(dev)].partition)
541                                             /* dev -> partition-index map I003 */
542     #define S5D_MINOR(unum,drnum,parnum) ((unum<<8)|(drnum<<4)|parnum)
543                                             /* I003 used in c215.c */
544
545
546     #define LHWORD(secnum)  (LOW(secnum),HIGH(secnum))
547                               /* I004 c order problem fix for user ease in c215.c */
548     #define                   LOW(x)            ((x)&0xFF)/* "low" byte */
549     #define                   HIGH(x)           (((x)>>8)&0xFF)/* "high" byte */
550
551     /* Gives offset and selector for a pointer   dab */
552     #define SELECTOR(x) ((unsigned int)(((long)(x))>>16))
553     #define OFFSET(x) ((unsigned int)((long)(x))&0xffff))
554
555     /* Whole disk partition table */
556     struct PCpart {
557     unsigned char bootind;
558     unsigned char bhead;
559     unsigned char b_psec;
560     unsigned char b_pcyl;
561     unsigned char sysind;
562     unsigned char ehead;
563     unsigned char e_psec;
564     unsigned char e_pcyl;
565     long relsec;
566     long numsec;
567     };
568
569     /*
570      * Winchester Drive Parameter Table
571     */
572
573     struct wparm_tab {
574             unsigned char cyls1;        /* number of cylinders */
575             unsigned char cyls2;
576             char heads;                 /* number of heads */
577             char write2_cur;            /* reduced write current */
578             char write1_cur;
579             char precmp1;               /* write precompensation */
580             char precmp2;
581             char ecc_len;               /* max. ecc burst length */
582             char control_byte;          /* enable retry, enable ecc, 70 usec steps */
583             char timeout;               /* standard timeout */
584             char fmt_timeout;           /* timeout for format drive */
585             char drvdiag_timeout;       /* timeout for test drive ready */
586             long zzj;
587             };
588
```

```
589     /*
590      * Floppy Drive Parameter Table
591     */
592
593     struct fparm_tab {
594             char spec1;                 /* first spec byte */
595             char spec2;                 /* second spec byte */
596             char optim;                 /* wait after opn til motor off */
597             char bps;                   /* bytes per sector */
598             char gap;                   /* gap length */
599             char dtl;                   /* DTL */
600             char gapformat;             /* gap length for format */
601             char fillbyte;              /* fill byte for format */
602             char hdsettle;              /* head settle time */
603             char motorstime;            /* motor start time */
604             };
605
606     /*
607      * Floppy Drive Parameter Table
608     */
609
610     /*
611      * Partition structure.  One per floppy drtab[] entry.
612      */
613
614     struct   PCd_part {
615             ushort                      p_flag;/* permission flag */
616             daddr_t                     p_fsec;/* first sector */
617             daddr_t                     p_nsec;/* number sectors */
618     };
619
620     /*
621      * VTOC structure for hard disk - one per wini drtab[] entry.
622      */
623
624     struct PCd_vtoc {
625             ushort v_sanity;            /* magic to verify vtoc */
626             ushort v_version;           /* layout version */
627             char v_volume[8];           /* volume name */
628             ushort v_sectorsz;          /* sector size */
629             ushort v_nparts;            /* number of partitions per volume */
630             unsigned long v_reserved[10];/* free space */
631             struct PCd_part pw[W_NPART];/* wini partitions */
632     };
633     /*
634      * Per-board configuration.
635      */
636
637     /*
```

```
638     * Per-board driver "dynamic" data.
639     */
640
641     struct    PCd_state {
642               char              s_active;         /* the state of the controller */
                                                      /* - IDLE or BUSY */
643               char              s_state;          /* what just finished (for interrupt) */
644               char              s_level;          /* what interrupt level (for PCd_io) */
645               char              s_flags;          /* flags per spindle; see below */
646               char              s_popen[OTYPCNT]; /* bit[i] ==> partition[i] open */
647               char              s_init;           /* status from init operation */
648               char              s_devcode;        /* device-code */
649               struct iobuf      *s_bufh;          /* -> buffer header */
650               unsigned          s_hcyl;           /* hold cylinder # during restore */
651     };
652
653     /*
654      * State Flags.
655      */
656
657     #define SF_OPEN        0x01     /* unit is open */
658     #define SF_READY       0x02     /* unit is ready; reset by media-change */
659     #define RESETTING      0x04     /* unit is resetting */
660     #define STATUSCK       0x08     /* checking status of disk operation */
661     #define INDIRECT       0x10     /* indirect disk operation (data copied */
                                        /* to/from low memory) */
662
663
664     /*
665      * Macros to make things easier to read/code/maintain/etc...
666      */
667
668     #defineIO_OP(bp)((bp->b_flags&B_READ) ? DSK_READ :
669     ((bp->b_flags&B_FORMAT) ? DSK_FORMAT : DSK_WRITE))
670
671     /* ALTERNATE TRACKING TABLE */
672     struct alt_tbl {                    /* needs to be defined*/
673               ushort    a_numbad;       /* number of bad tracks        */
674               ushort    a_fstalt;       /* first track of alternate area*/
675               ushort    a_lstalt;       /* last track of alternate area */
676               ushort    a_maxbad;       /* total number of allowable bad tracks*/
677               struct alt {
678                    ushort       a_btrk;      /* packed bad track    */
679                    ushort       a_gtrk;      /* packed good track   */
680                    } a_alt[MAXBAD];
681     };
682
683     struct    PCd_drtab {
684               unsigned          dr_ncyl;     /* # cylinders *
```

```
685             char            dr_nhead;       /* # heads */
686             char            dr_nsec;        /* # sectors per track */
687             struct alt_tbl *dr_altptr;      /* alternate track table pointer */
688                                             /* if floppy, 0==FM, 1==MFM */
689             unsigned        dr_spc;         /* actual sectors/cylinder */
690             unsigned        dr_spb;         /* sectors/block */
691             unsigned        dr_secsiz;      /* sector-size (bytes) */
692             struct PCd_part *dr_part;       /* partition table pointer */
693     };
694
695     struct  PCd_cdrt {
696             unsigned        cdr_ncyl;       /* # cylinders */
697             char            cdr_nhead;      /* # heads */
698             char            cdr_nsec;       /* # sectors per track */
699             unsigned        cdr_secsiz;     /* sector-size */
700             struct PCd_part *cdr_part;      /* partition table pointer */
701     };
702
703     /*
704      * Device-Data.  One per board (declared in driver).
705      */
706
707     struct  PCd_dev {
708             struct          PCd_state       d_state;
709             struct          PCd_drtab       d_drtab;
710     };
711
712     /*
713      * Values of buffer-header b_active, used for mutual-exclusion of
714      * opens and other IO requests.
715      */
716
717     #define  IO_IDLE            0           /* idle -- anything goes */
718     #define  IO_OPEN_WAIT       1           /* open waiting */
719     #define  IO_BUSY            2           /* something going on */
720     #define  IOC_WAIT           3           /* waiting for the device */
721
722     /*
723      * Values of PCd_state.s_devcode, internal driver state.
724      */
725     #define FLPY            0x00            /* BIOS floppy disk selector */
726     #define WINI            0x00            /* BIOS winchester selector */
727
728     /*
729      * Floppy FM/MFM codes for drtab[*].nalt.
730      */
731
732     #define  FLPY_FM        0               /* FM -- single density */
733     #define  FLPY_MFM       1               /* MFM -- double density */
```

```
734
735     #define  FDESCR        8           /* Floppy workspace descriptor */
736     #define  WDESCR        9           /* Wini workspace descriptor */
737
738     #define  NWCONFIG      32          /* number of Wini parameter tables */
739
740     /*
741      ******************************************************************************
742       * Parameters common to Fdisk.c and Format.c in regard to the whole disk
743       * partition table
744      ******************************************************************************
745       */
746
747     #define PARENT    4            /* Number of entries within the partition table */
748                               /* maximum four entries on whole disk */
749     #define UNIXOS    99       /* UNIX + DOS (merged) partition */
750     #define DOSOS     1        /* DOS only partition */
751     #define DOSOS16   4        /* DOS only partition (16-bit FAT) */
752     #define DOSDATA   86       /* DOS-DATA partition */
753     #define ACTIVE    128      /* Current partition is active (only 1 per */
                                   /* drive is allowed) */
754     #define EMPTY     100      /* No partition (partition slot unoccupied) */
755     #define MIN_USIZ  19       /* Minimum size (cylinders) for UNIX partition */
756     #define MAXDOS    65535L   /* max size (sectors) for a DOS partition */
757
758     /******************************************************************************/
```

E Appendix E

A Sample Driver Software Package

This appendix contains the ID modules needed to install a device driver. Most of the intelligence of a driver package is in the **Install** and **Remove** scripts. In particular, the **Install** script should test if the driver package is already installed and advise the user accordingly. The **Install** script can also interrogate which interrupt vectors and/or I/O addresses are available on the system via the **idcheck** command and provide an interactive script with which the user can select an available interrupt or address and set appropriate straps on the controller board.

The driver package for the trace driver described in Appendix C is provided here as an example. Although this package does not use interrupts or an I/O address (as seen in the System module), most of the content of this driver package relates to any device driver.

The **Install** script presented here contains a large amount of diagnostic and recovery information such as checking if the package is already installed and overwriting the old package if the user confirms the message(1) program. The script also gives the user a chance to examine reconfiguration errors if the build fails by redirecting errors to a file in **/tmp.** If the installation proceeds without errors, the user never sees any of this. It is up to the driver writer to decide what level of error recovery and user diagnostics are needed. In the simplest case, the **Install** script needs only to call two commands: **idinstall** and **idbuild**. Some items to note in the **Install** script:

- Make liberal use of the **echo** and **message** commands to tell the user what is going on.
- Make sure you exit with a non-zero return code on installation errors.
- Clean up after an aborted installation.
- In the example (lines 40-43), note that the driver header file is installed before the system reconfiguration. This is done because the reconfiguration process requires the header file to be in **/usr/include/sys**. The remaining user files and commands are installed (lines 55-72) after a successful reconfiguration. This method makes cleanup after an aborted installation a bit easier.

```
1       #
2       # 386unix Sample Driver Install Script
3       #
4
5       TMP=/tmp/trace.err
6       ERROR1=" Errors have written to the file $TMP."
7       ERROR2=" The Trace Driver software was not installed, and the System
8       has not been modified.
9       Please call the Trace Software Hotline if you
10      need additional information."
11
12      echo "Installing Trace Device Driver Software Package"
13      /etc/conf/bin/idcheck -p trace 2> $TMP
14      if [ $? != 0 ]
15      then
16              message -cu "The Trace Driver is already installed (or
17                      partially installed).
18              Do you wish to overwrite the existing device
19                      driver software?"
20              if [ $? = 0 ]
21              then
22                      /etc/conf/bin/idinstall -d trace
23              else
24                      exit 1
25              fi
26      fi
27
28      /etc/conf/bin/idinstall -a trace 2>> $TMP
29      if [ $? != 0 ]
30      then
31              message "There was an error during package installation.  ERROR1 $ERROR2"
32              exit 1
33      fi
34
35      #
36      # Install the .h file before the build; Space.c needs a structure
37      # definition contained therein.
38      #
39
40      mv trace.h /usr/include/sys
41      chown bin /usr/include/sys/trace.h
42      chgrp bin /usr/include/sys/trace.h
43      chmod 444 /usr/include/sys/trace.h
44
45      /etc/conf/bin/idbuild 2>> $TMP
46      if [ $? != 0 ]
47      then
```

```
48              # If an error occurs here, remove the driver components.
49              /etc/conf/bin/idinstall -d trace
50              rm -rf /usr/include/sys/trace.h
51              message "There was an error during Kernel reconfiguration. $ERROR1 $ERROR2"
52              exit 1
53      fi
54
55      echo "Installing Commands"
56      for ii in trfmt trsav
57      do
58              mv $ii /usr/bin
59              chown bin /usr/bin/$ii
60              chgrp bin /usr/bin/$ii
61              chmod 755 /usr/bin/$ii
62      done
63      for ii in trace.c trfmt.c trsav.c
64      do
65              mv $ii /usr/src
66              chown bin /usr/src/$ii
67              chgrp bin /usr/src/$ii
68              chmod 444 /usr/src/$ii
69      done
70
71      rm -f $TMP
72      exit 0
```

```
1       #
2       # 386unix Sample Driver Remove Script
3       #
4       #
5
6       TMP=/tmp/trace.err
7       RERR="An error was encountered removing the Trace Driver Package.
8               The file $TMP contains errors reported by the system."
9
10      echo "Removing Trace Device Driver Software Package"
11      /etc/conf/bin/idinstall -d trace 2> $TMP
12      if [ $? != 0 ]
13      then
14              message $RERR
15              exit 1
16      fi
17
18      /etc/conf/bin/idbuild 2>> $TMP
19      if [ $? != 0 ]
20      then
21              message $RERR
22              exit 1
23      fi
24
25      echo "Removing Commands"
26      rm -f /usr/include/sys/trace.h /usr/bin/trsav /usr/bin/trfmt
27      rm -f /usr/src/trace.c /usr/src/trsav.c /usr/src/trfmt.c
28
29      rm -f $TMP
30      exit 0
```

The Name file:

```
386unix Trace Device Driver Package
```

The Size file:

```
ROOT=1500
USR=500
```

The Files file:

```
/usr/include/sys/trace.h
/usr/bin/trsav
/usr/bin/trfmt
/usr/src/trace.c
/usr/src/trsav.c
/usr/src/trfmt.c
```

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
```

F Appendix F

Porting

Common Utilities

The following utilities are required to exist on UNIX System V/386 machines, to have the same functionality and the same full path name, and as such can be used in portable programs.

/bin

cat	df	login	ps	sort
chgrp	du	ls	pwd	stty
chmod	echo	mail	red	su
chown	ed	mesg	rm	sync
cmp	expr	mkdir	rmail	true
cp	false	mv	rmdir	uname
cpio	grep	od	rsh	wc
date	kill	passwd	sh	who
dd	ln	pr	sleep	write

/usr/bin

getopt news tabs

/etc

brc	getty	mknod	rc	umount
devnm	killall	mount	shutdown	wall
fsck	mkfs			

/usr/lib

makekey

The **mail, mesg, news, wall, who,** and **write** commands are specified here but do not exist in the UNIX XS utilities list. The **bcheckrc, clri, config, glossary, help, init, labelit, locate, startup, shutdown, telinit,** and **usage** commands are specified in the XS list but not here.

Common System Calls

The following system calls are required to exist on all UNIX System V/386 machines and to have the same functionality, and as such can be used in portable programs.

_exit	execlp	getppid	plock	time
access	execv	getuid	profil	times
alarm	execve	ioctl	ptrace	ulimit
brk	execvp	kill	read	umask
chdir	exit	link	sbrk	umount
chmod	fcntl	lseek	setgid	uname
chown	fork	mknod	setpgrp	unlink
chroot	fstat	mount	setuid	ustat
close	getegid	nice	signal	utime
creat	geteuid	open	stat	wait
dup	getgid	pause	stime	write
execl	getpgrp	pipe	sync	
execle	getpid			

This list contains all the system calls found in the *Programmer's Reference Manual* with the exception of *acct*(2), the *message* system calls, the *semaphore* system calls, and the *shared memory* system calls.

The **_exit, setgid,** and the **sys3b** system calls are specified here but do not exist in the UNIX XS system call list.

Common Signal Set

The following list defines the set of signals that are available for use with the *signal(2)* system call and are common to all UNIX System V/386 machines, and therefore portable.

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03	quit
SIGILL	04	illegal instruction (not reset when caught)
SIGTRAP	05	trace trap (not reset when caught)
SIGIOT	06	IOT instruction
SIGEMT	07	EMT instruction
SIGFPE	08	floating-point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	death of a child
SIGPWR	19	power fail

SYSI86 Argument Set

The following list defines the set of arguments that are available for use with the **sysi86** system call.

SI86SWPI	1	General SWAP interface
SYSI86DMM	7	Double-map data segment for read/write/executive support
GRNON	52	Set green light to solid on state
GRNFLASH	53	Start green light flashing
STIME	54	Set internal time
SETNAME	56	Rename the system
RNVR	58	Read NVRAM
WNVR	59	Write NVRAM
RTODC	60	Read time of day clock
CHKSER	61	Check soft serial number
SI86NVPRT	62	Print an xtra_nvr structure
SANUPD	63	Sanity update of kernel buffers

Index

DOC0041-2Y