VENIX/11

INSTALLATION AND SYSTEM MANAGER'S GUIDE

VENTURCOM

VenturCom, Inc., 215 First Street, Cambridge, MA 02142, 617/661-1230

BOOT 11/70: HALT DOWN, PRESS START A FEW TIMES TO RESET, 17765744_8 IN SWITCHES (1111 111 110 101 111 100 100 1=UP) LOAD-ADDR, 50_8 IN SWITCHES (0 —— 000 101 000 1=UP), HALT UP, START, & venix

VENIX/11 Installation and System Manager's Guide

BOOT 11/34: CTRL-HALT, CTRL-BOOT

→ @ nn

nn = DL	RL02
DB	RM05
MT	KENNEDY TAPE

→ & venix

VENIX† is derived from UNIX‡.

PROBLEMS: • SWAP ERROR	DISK WRITE PROTECTED
• "Venix 2.0 ..." "PPP...... PP"	CONSOLE LINE HAS INCORRECT VECTOR (60_8)

September 1984

The VENIX† Documentation Set

The VENIX documentation set consists of the following manuals:

VENIX Installation and System Manager's Guide

The set up and maintenance of VENIX are described in the installation sections. Other articles explain the UNIX-to-UNIX‡ communications systems. The "System Maintenance Reference Manual" contains reference pages for devices and system maintenance procedures (sections (7) and (8)).

VENIX User Guide

The *User Guide* contains tutorials for newcomers to VENIX, covering basic use of the system, the editor **vi** and use of the command language interpreters.

VENIX Document Processing Guide

The line and screen editors and **nroff**-related text formatting utilities are described in the Document Processing Guide. Topics include: line editor **ed**, and stream editor **sed**; the text formatter **nroff**; the **nroff**-preprocessors **tbl** and **neqn**.

VENIX Programming Guide

The chapters in the *Programming Guide* explain the different programming languages for VENIX.

VENIX Support Tools Guide

This guide includes tools for programming, such as the compiler-writing languages Yacc and Lex, the M4 Macro processor, the program development utility Make, and the desk calculator programs DC and BC.

VENIX User Reference Manual

This is a complete and concise reference for the VENIX system. This volume contains write-ups on all VENIX commands.

VENIX Progammer Reference Manual

The reference pages in this volume include system calls, library functions, file formats, miscellaneous functions and games.

Contents

Introduction

The *Installation and System Manager's Guide* covers the installation and maintenance of the VENIX/11 operating system, and implementation of the VENIX communications system. The following paragraphs contain a brief description of each chapter.

The first chapter, SETTING UP VENIX/11, explains how to install the VENIX/11 operating system on your hardware.

The chapter VENIX MAINTENANCE is a guide to VENIX administration, maintaining file systems, managing disk space, backing up user files and recovering and diagnosing system errors.

The chapter UUCP IMPLEMENTATION DESCRIPTION explains how to install and administer your UUCP system.

The chapter A DIAL-UP NETWORK OF UNIX SYSTEMS describes a network that provides information exchange between UNIX systems over the direct distance dialing network.

The chapter THE VENIX I/O SYSTEM describes the workings of the I/O system with particular emphasis on the environment and nature of device drivers.

The SYSTEM MAINTENANCE REFERENCE MANUAL contains reference pages for devices and system maintenance procedures (sections (7) (8)).

A few words on notation: throughout this manual, frequent citations are made to pages in the *User Reference Manual* or *Programmer Reference Manual*. These are often in the form of a name followed by a section number, e.g. **fsck(1)**, referring to the **fsck** command description in section one of the *User Reference Manual*.

Commands to be typed in literally by the user are given in **bold**; generic arguments are in *italics*.

Chapter 1

SETTING UP VENIX/11

1.1 HOW TO USE THIS DOCUMENT

This document contains-step-by-step instructions for installing the VENIX/11 system. If you don't have previous UNIX/VENIX experience, look through "VENIX for Beginners" in the *User Guide* before plunging into the installation. You should at least be able to do simple editing and move around in the file system.

A few words on notation: throughout this document, frequent references will be made to pages in the *User Reference Guide*. These will always be in the form of a name followed by a section number, e.g. **fsck**(1), referring to the **fsck** command description in section one of the *User Reference Manual*.

Examples of commands you should type are printed in **bold**; responses from VENIX are in plain text. Commands are almost always printed in lower-case letters.

After you copy the system onto hard disk, you will need to customize it to your particular installation by editing tables and files. The **ed** editor is a line-oriented editor which will work on any terminal, and you may find this the easiest to use at first. You could also use the **ice** (VT-52/VT-100 only), **vi**, or **fw** (The FinalWord) screen editors to make these changes (see **ice**(1) and **vi**(1)). After the system is up and going, you will probably find the screen editors preferable for day-to-day use.

1.2 CREATING THE SYSTEM IMAGE

1.2.1 Minimum Hardware Required

VENIX requires a PDP- or LSI-11 computer with at least 192k bytes of memory, memory management, a console terminal, a line clock, a mass storage disk (system device), and a device to read the distribution media if different from the system disk. Floating point hardware is not generally required, but it is recommended if any amount of floating point computation is to be done; it is needed for the Fortran 77 compiler **f77**, the Pascal compiler/interpreter **pc**, and the C verifier **lint**. If you are using removable disk packs, error-free packs are strongly recommended, especially for the system disk.

VENIX requires a minimum of 8mb of disk space. As distributed, it will run from disk unit 'zero' (or units 'zero' and 'one' for RL01-based systems). After VENIX is installed, it may be reconfigured for different disk units.

The preconfigured VENIX kernel supplied with the VENIX/11 distribution is limited to 256kb of memory, so that it will run on machines with or without 22-bit addressing capabilities. The user may reconfigure the kernel to take advantage of additional memory on the system, providing the above restrictions are heeded.

1.2.2 Preliminary Hardware Configuration

The devices used in the installation (floppy or magtape and hard disk) should all be set to standard interrupt and register addresses, and standard priorities. The memory should, of course, be configured contiguously. The hard disk must be write-enabled so that a system can be copied onto it. (On RL01-based systems, both units 'zero' and 'one' must be write-enabled and ready.) Devices which are not used during the installation itself may be set in a nonstandard manner, since you can accommodate these differences in software once the system is up and going (as described in "Adding Drivers").

The DLV-11J serial interface board for LSI-11/23 machines has a 'reboot on break' option, selected by an on-board jumper, which will halt the computer when the break key is pressed. This feature should be disabled when running VENIX, as suddenly halting VENIX will lead to corrupted file systems and other miseries. If VENIX is accidentally halted in this way, it should be immediately resumed.

1.2.3 Distribution Media

VENIX is distributed in one of three ways:

1. Bootable XFER diskette plus SYSTEM and USER AREA diskettes. (RX50 or RX02-compatible)
2. Bootable SYSTEM AREA magnetic tape (800 bpi TM11-compatible), with a separate USER AREA tape.
3. Bootable hard disk (RL02).

In the first two cases, some work must be done to transfer the VENIX image from floppies or magtape to the system disk where it will run. For floppy transfers, continue to the next section; for tape, refer to "Transferring from Magtape". If you received the VENIX system on RL02 disk, then the system is already installed. Load the disk, and proceed directly with "Some Initialization Exercises".

1.2.4 Transferring from Floppy

1.2.4.1 Booting the XFER Floppy

The floppy diskette distribution contains a small runnable VENIX system which can be booted directly from the PDP-11 console. The XFER diskette, which must be write-enabled, should be placed in drive 0. (All other diskettes should be write-protected). At this point, the hard disk drive unit 0 must be ready. (Both units 0 and 1 must be ready for RL01 systems). Follow the instructions in the hardware manual for booting the floppy.

An '&' prompt should appear on the console terminal. Type

venix

followed by a CR to bring the small system up. (Typing errors can not be corrected while entering the name 'venix', but if an illegal name is given, you will receive another '&' prompt.)

After a minute or so, a 'VENIX' message will appear on the screen, followed by a serial number and memory size in kilobytes. After some introductory messages you will be asked:

Do you wish to install VENIX (y or n)?

Type 'y'.

If you type 'n' the shell will be executed, and you will be able to run VENIX commands. (This may be useful in the future if you are experienced with VENIX and wish to run maintenance programs not possible through XFER. To restart the transfer process, type CTRL-D to log out.)

If you can't get this far, or experience later difficulties in copying from floppies, then see the section "Difficulties in Booting".

1.2.4.2 PART I — Selecting the Disk

At this point you will be presented with a list of hard disk types, and asked which one is your system disk. After you type the number corresponding to your disk, you will be asked if you have one or two floppy drives. Type '1' or '2'. If you have two floppy drives, type '2' and skip to the "User Area" section in Part II. Otherwise, type '1' and continue to the next section.

1.2.4.3 Secondary Boot, for Single-Floppy Systems Only

If you have a single-floppy system, XFER will now copy an image of the XFER floppy onto a temporary area of the hard disk. This step is necessary because you can not simultaneously run the system and swap in and out the USER and SYSTEM AREA diskettes, if you only have one floppy drive. Therefore, you have to run a small system from the hard disk, leaving the floppy drive free for the USER and SYSTEM AREA diskettes.

After copying itself to hard disk, XFER will stop and instruct you to reboot. Leave the XFER diskette in the drive, halt the processor and reboot the bootable floppy exactly as you did before. This time, when you receive the '&' prompt, type

venix.tmp

instead of venix. Even though you have booted from the floppy drive, this version of the system will actually begin running from the hard disk. XFER will begin running again. After asking your disk type, it will be ready to install the USER AREA diskettes.

1.2.4.4 PART II — USER AREA

The next direction on the screen will be:

Part II — Restoring the User Area
Do you wish to restore the user area? (y or n)

Type 'y'. (If you type 'n' XFER will skip directly to Part III, installing the system area.)

At this point, XFER checks to see if there are any VENIX files in the user area of the disk from a previous installation. A warning will appear automatically. It will say 'WARNING: There is a VENIX file system already on the user area. Do you wish to continue (y or n)?' If you type 'n', VENIX will skip to Part III, leaving the user area intact. If you type 'y', the user area will be erased, and the transfer process will continue.

The next question to appear on the screen is:

Do you wish to check for bad blocks on the user area (y or n)?

If you are sure that the disk is error-free, type 'n'. Otherwise, type 'y'.

A file system will now be constructed on the user area; bad block checking is done at the same time if it was requested. The command shown will be something like

```
/etc/mkfs -b /dev/rl0.usr 10240
```

This will take about five minutes if bad block checking was requested, or about fifteen seconds if the checking was not chosen. (In the latter case the, **−b** flag will not be given in the above command). The number 10240 indicates the number of blocks in the user area, and will vary depending on the size of the hard disk.

If a bad block is found on the hard disk, you will receive an error message of the form 'Err on dev X/Y ...' followed by the message 'Bad block number *n*'. VENIX can tolerate up to eight bad blocks in the file system it creates. In the event that more than eight bad blocks are found, the message 'Too many bad blocks' will be given. If this message is given, or the message 'Bad block in i-node region' appears, then the hard disk will be unusable with VENIX. The only recourse if this happens is to get a new disk and retry the installation procedure.

The user area holds some system command and library files. In this next section, these files will be loaded onto the hard disk.

You will next receive the message:

Ready to transfer files from USER AREA diskettes.

If you are running on a double-drive floppy system, the XFER diskette will remain in drive zero for the entire procedure. On single-drive floppy systems, the XFER diskette should now be removed from the drive. The USER AREA diskettes will be swapped in and out of the free drive.

You will now receive the message:

Insert USER AREA diskette #1 and press 'return' ...

After you have placed the USER AREA diskette in the free drive, press the CR key. The **tar** command will be run to transfer the user files from the diskette to the winchester. You will see something like:

tar xf /dev/rrx1 /usr

Starting ...

After the loading has finished, XFER will print on the screen:

Insert USER AREA diskette #2 and press 'return' ...

Remove USER AREA diskette #1 from the drive and insert diskette #2. Press

CR. You will see the **tar** command being executed again and the word 'Starting ...' appear on the screen.

When diskette #2 is finished, you will be asked to insert diskette #3, and so on, for all the diskettes. In each instance you will be told when to take out the diskette and insert the next one in sequence.

When all the USER AREA diskettes have been transferred, Part II is completed. The message

User area completed.

is displayed. Withdraw the last USER AREA diskette from the drive.

1.2.4.5 Recoverable Errors

If, during this loading procedure, you do any of the following things:

- Insert a diskette in the drive upside down;
- Press CR when no diskette is in the drive;
- Remove the XFER diskette from drive 0;

you will receive HARD or SOFT error messages.

1. The error message 'HARD ERROR in reading this diskette' means that the diskette is inserted improperly, or has a defective surface.

2. The error message 'SOFT ERROR in reading this diskette' means that the wrong diskette was inserted.

In both cases, after the error message, you will be asked:

```
Would you like to
        Abort the transfer
        Retry reading the diskette, or
        Ignore the error?
Type 'a', 'r', or 'i'.
```

Typing ‘a’ causes the transfer to abort. Power off and start again. If you have mistakenly removed the XFER diskette from drive 0, you should always make this choice.

Typing ‘r’ allows you to retry reading the user area diskette. You will be instructed to:

Check diskette and press CR when ready.

You can remove the diskette from the lower or right-hand drive and check to make sure it is the right one. If it is the wrong diskette, reinsert the right diskette, and press CR.

Typing ‘i’ causes the error to be ignored. The system will be installed, but one or more files may be missing or corrupted.

If you repeatedly get a HARD ERROR, the diskette may be bad. Contact your VENIX distributor. If you suspect you have hardware problems, see the section “Difficulties in Booting”.

XFER will let you insert the USER AREA diskettes in the wrong order, provided that you do not attempt to use the same diskette twice in one session. But since inserting the diskettes out of order may lead to minor permission problems on the winchester user area, it is strongly recommended that you stick to the order given by the diskette labels.

1.2.4.6 PART III — SYSTEM AREA

You have just completed loading the user area onto the winchester hard disk. In this section, Part III, the system area will be transferred to the hard disk. The procedure is nearly identical to that used for the user area.

Now, the screen will say:

Part III — Restoring the System Area

Do you wish to restore the system area (y or n)?

Type ‘y’. (If you type ‘n’, XFER will go to the end of the loading process. This should only be done if you have previously installed a system area on the disk).

If a VENIX file system was found on the system area — left over from a previous installation — a warning will appear (‘WARNING: There is a VENIX file system already on the system area. Do you wish to continue (y or n)?’). If you type ‘n’, XFER will preserve the files and halt the installation; if you type ‘y’, the files in the system area will be obliterated.

The warning mentioned above will not occur if you are loading VENIX for the first time. You will now see:

Do you wish to check for bad blocks on the system area (y or n)?

If you are sure that the disk is error-free, type in ‘n’. Otherwise, type ‘y’. NOTE: it is strongly recommended that system disks be error-free.

XFER will now make a file system and check for bad blocks on the hard disk. The command executed will be:

/etc/mkfs −b /dev/*XX*.sys 6816

If bad block checking was not requested, the **−b** is left out of the command shown above. The same rules apply to bad block checking on the system area as given before for the user area: up to eight bad blocks can be handled successfully, but if more than eight bad blocks are found, or a bad block is listed in the ‘i-node region,’ then the winchester will have to be replaced for successful installation.

The next step is loading the primary system area (‘root’ area) onto the hard disk. VENIX first tells you that the system will now be transferred onto the hard disk, and then prompts you to insert the first system area diskette.

Ready to transfer files from SYSTEM AREA diskettes.

Insert SYSTEM AREA diskette A and press ‘return’ ...

Insert SYSTEM AREA diskette A in the free drive. When ready, press CR. The **tar** command will be run to transfer the system area files from diskette to winchester, looking something like:

```
tar xf /dev/rrx1 .

Starting ...
```

When the loading is finished, you will be instructed to:

```
Insert SYSTEM AREA floppy B and press 'return' ...
```

Remove SYSTEM diskette A and insert diskette B. Press CR. You will see **tar** being executed and the word 'Starting ...' appear again.

When diskette B is finished, you will be asked to insert diskette C, and so on, for all four diskettes. In each instance, you will be told when to take out the diskette and insert the next one in sequence.

If you get a HARD or SOFT ERROR message during the loading procedure, see section "Recoverable Errors".

When all the SYSTEM diskettes have been transferred, you are finished with XFER. Halt the processor, remove the distribution medium, and proceed with "Some Initialization Exercises".

1.2.5 Transferring from Magtape

If you are transferring the system from magtape, the target hard disk which you intend to run your system from must be error-free, formatted, and loaded in drive 0. The bootable magtape, labeled 'BOOTABLE SYSTEM AREA', should be mounted on tape drive 0 and positioned at the load point. (The Line Time Clock (LTC) must be disabled now if your machine is an LSI-11/23.) Key in and execute the following small program at location 0100000 (octal):

```
012700
172526
010040
012740
060003
000777
```

After this program is executed, the tape should move and the CPU should loop. Halt the processor and restart at location 0.

When you have done this, an '=' prompt should appear on the console terminal. Your options are as follows:

Type:	*If your disk device is:*
rl	**RL01 or RL02**
rk06	**RK06 or RK07**
rm02	**RM02 or RM03**
rm04	**RM04 or RM05**
rp02	**RP02 or RP03**
rp04	**RP04, RP05 or RP06**

The disk name must be in lower-case, and must be followed by a CR.

The tape should now move and the transfer begin.

After the transfer is finished, the tape will rewind and the CPU will halt. If no problems occur, then skip to the section "Some Initialization Exercises".

1.2.6 Difficulties in Booting

1.2.6.1 General

There are several types of problems which will prevent you from booting the floppy or magtape. If no prompt is sent to the console, then you may have a problem with the terminal. Check that it is on the console line, and that the the baud rates of the terminal and the DL match.

If the floppy or magtape device doesn't do anything, or appears to hang, then it may be that the device interrupt vectors or registers are not at standard addresses. These addresses should be checked against the hardware manual. The computer console switches (or ODT on LSI-11's) can be used to verify the existence of registers at the appropriate addresses. And of course if the memory is addressed incorrectly, all sorts of difficulties could arise; this can also be checked out with the console switches or ODT, by attempting to read various locations in memory.

The line-time clock (LTC) must be disabled when booting from magtape, and it wouldn't hurt to disable it when booting floppies as well. It should be enabled when the full VENIX system is up and running.

The hard disk you are copying to must, of course, be write-enabled.

Another possible source of problems is other devices on the backplane, which could be interfering with interrupts or otherwise corrupting operations. If nothing else seems to work, remove all non-essential devices from the backplane.

1.2.6.2 Floppies

If the floppy does not prompt you with an '&', it may be that the system diskette is unreadable by the floppy drive for one reason or another. If you have been running RT-11 or some other operating system, you can check if the diskette is readable by attempting to read raw data from the floppy. (The DUMP utility under RT-11 will let you try this). If this existing operating system finds the diskette totally unreadable, then you should consult your distributor for further instructions.

The memory size (in kbytes) given on booting represents the amount of memory VENIX thinks is available for user programs. This should be at least 140 kb, and will be larger for systems with more than the minimum 192 kb of memory.

An error message of the form 'Error ...' indicates an error in reading or writing to either the floppy itself or the hard disk. This could be due to incorrectly configured hardware, a misaligned floppy drive, or write protection on either the system floppy or the hard disk (both must be write-enabled). A message beginning with 'PANIC ...' indicates an unrecoverable error, and will be followed by silence. In either case, record the exact error codes. If problems persist after several attempts, consult your distributor for further advice.

If your system hangs without any error message while copying from floppy diskette to hard disk, the priorities on either the floppy or hard disk controller may be incorrect. They both should be set to five.

1.2.6.3 Magtape

The distribution magtape is at 800 bpi, which must match the setting on the drive. It is worth rechecking that the bootstrap was entered correctly, and placed at the correct location.

1.3 SOME INITIALIZATION EXERCISES

1.3.1 Booting

At this point, it is assumed that your VENIX system is now on your main system disk, having been either transferred successfully from floppy or magtape, or distributed on the hard disk itself. Your full VENIX system is now ready to be booted. Boot block zero of the *hard disk* from your console. You will receive an '&' prompt.

The following kernels are preconfigured and may be booted at this point:

Name	*Target Disk*
rdvenix	**RD51 (Micro/PDP-11 winchester)**
rlvenix	**RL01 or RL02**
rkvenix	**RK06 or RK07**
rm2venix	**RM02 or RM03**
rm5venix	**RM05**
rp2venix	**RP02 or RP03**
rp4venix	**RP04, RP05 or RP06**
smsvenix	**Scientific Micro Systems FWT**

After the '&' prompt, type the name of the kernel corresponding to your system disk followed by CR, to bring up your system. (Typing errors can not be corrected while entering the name *XX***venix** but if an unrecognized name is given, you will receive another '&' prompt.)

The system will print a 'VENIX' message on the console terminal, as well as a serial number and memory size. The serial number is a unique number assigned your particular system. The memory size is the amount of memory available for user programs (in kilobytes), equal to the total memory available minus the VENIX kernel size; the latter will be 40-48kb. At this point, only 256 kb of memory will be recognized on the system, so the maximum available memory will be about 210 kb.

You may also receive a message-of-the-day — something inspiring from your distributor. (If this becomes monotonous on future logins, the message may be found and deleted in file **/etc/motd**.) Finally you will receive the message

```
Beware: you are a super user!
SUPER>
```

VENIX is now up and running in single-user mode. This means that the console is the only active terminal. The 'Beware ...' message is simply a warning that you are the 'super-user' while in single-user mode, and have unrestricted access to the system. The 'SUPER>' prompt is a repeated reminder of your status.

Normal VENIX commands may be given, but it is better to proceed with the installation and go into multi-user mode before doing too much. Certain commands are not loaded yet, and will not be found if you try to execute them.

As long as you remain in single-user mode, you can always turn off the computer safely provided you

1. Stop any programs that you may be running;
2. Unmount any mounted file systems you may have mounted (described later); and
3. Type the command **sync**.

The method for shutting down VENIX from multi-user mode is described later.

1.3.2 What Hardware is Accessible?

Your VENIX system has been preconfigured to handle a single type of hard disk (one or two units), a pair of floppy drives, one tape drive, and four serial lines. The system has a preconfigured limit of 256kb memory, of which approximately 210 kb is available for user programs.

The following table shows the number of units supported for each disk type, and the logical VENIX partitions ("areas") for each unit. One of the following disk types is supported in your preconfigured kernel.

64966

65056

Disk	Unit	Name	Description	Blocks				
RD51	0	/dev/w0.sys	system area	6816				
		/dev/w0.tmp	temp area	2400				
		/dev/w0.usr0	1st user area	10240				
RK06	0	/dev/rk0.sys	system area	6816				
	″	/dev/rk0.usr	1st user area	16830				
	1	/dev/rk1.usr	2nd user area	27126				
RK07	0	/dev/rk0.sys	system area	6816				
	″	/dev/rk0.usr	1st user area	43494				
	1	/dev/rk1.usr	2nd user area	53790				
RL01	0	/dev/rl0.sys	system area	6816				
	″	/dev/rl0.tmp	temp area	2400				
	1	/dev/rl1.usr	1st user area	10240				
RL02	0	/dev/rl0.sys	system area	6816				
	″	/dev/rl0.tmp	temp area	2400				
	″	/dev/rl0.usr	1st user area	10240				
	1	/dev/rl1.usr	2nd user area	20480				
RM02/03	0	/dev/rm0.boot	boot area	160				
	″	/dev/rm0.usr0	1st user area	50880				
	″	/dev/rm0.sys	system area	6816				
	″	/dev/rm0.tmp	temp area	2400				
	″	/dev/rm0.usr1	2nd user area	50880				
	″	/dev/rm0.usr2	3rd user area	19520				
RM05	0	/dev/rm0.boot	boot area	608	0	16	RM	BOOT
	″	/dev/rm0.usr0	1st user area	65056	1	17		0
	″	/dev/rm0.sys	system area	6816	2	18		SYS
	″	/dev/rm0.tmp	temp area	3648	3	19		TMP
	″	/dev/rm0.usr1	2nd user area	65056	4	20		1
	″	/dev/rm0.usr2	3rd user area	65056	5	21		2
	″	/dev/rm0.usr3	4th user area	65056	6	22		3
	″	/dev/rm0.usr4	5th user area	65056	7	23		4
	″	/dev/rm0.usr5	6th user area	65056	8	24		5
	″	/dev/rm0.usr6	7th user area	65056	9	25		6
	″	/dev/rm0.usr7	8th user area	31008	10	26		7
RP02	0	/dev/rp0.sys	system area	6816				
	″	/dev/rp0.tmp	temp area	2400				
	″	/dev/rp0.usr0	1st user area	30200				
	1	/dev/rp1.usr0	2nd user area	40600				
RP03	0	/dev/rp0.sys	system area	6816				
	″	/dev/rp0.tmp	temp area	2400				
	″	/dev/rp0.usr0	1st user area	30200				
	″	/dev/rp0.usr1	2nd user area	40600				
	1	/dev/rp1.usr0	3rd user area	40600				
	″	/dev/rp1.usr1	4th user area	40600				
SMS	0	/dev/sms.sys	system area	6816				
	″	/dev/sms.tmp	temp area	2400				
	″	/dev/sms.usr	1st user area	10240				

9

Standard register and interrupt vector addresses are used for all the above disks.

Your system has also been preconfigured for four DL-11 serial lines, at the following register addresses and interrupt vectors:

Channel	Address	IVA	Name
0	0176500	0300	/dev/tty00
1	0176510	0310	/dev/tty01
2	0176520	0320	/dev/tty02
3	0177560	060	/dev/console

One set of floppy drives is also supported, at standard register and interrupt vector addresses.

Device	Unit	Name	Blocks	Description
RX02	0	/dev/rx0	988	Drive 0, double density
	0	/dev/rx0s	494	Drive 0, single density
	1	/dev/rx1	988	Drive 1, double density
	1	/dev/rx1s	494	Drive 1, single density
RX50	0	/dev/f0	790	Drive 0
	1	/dev/f1	790	Drive 1

The RX02 drives are supported on all distributions except for Micro/PDP-11 distributions. RX50 drives are supported on the Micro/PDP-11. The preconfigured drivers will support double-sided double-density operation on RX02 drives which have this feature, in which case 1988 blocks may be accessed.

The system is also preconfigured for one TM-11 or TU-10 tape drive, at the standard register and interrupt vector address.

Name	Description
/dev/mt0	Drive 0, standard
/dev/nmt0	Drive 0, no-rewind on close

All of the above devices are known by two names. The standard name, given above, is the device's 'block' name. Each of these names has a 'raw' counterpart, which is sometimes used to access the device in a faster, more direct way.

These raw names are formed by prepending an 'r' to the original name, to form for example '/dev/rrx0' for the raw version of floppy drive 0, or '/dev/rrl0.sys' for the raw version of the RL02 system area. The raw version of a device can not be used with all programs, but will be shown where valid in examples later in this document. (NOTE: 'raw' entries should not be used for the first user partition on RM-type drives, since it is inverted.)

A complete list of all the device nodes preconfigured in your system can be obtained by using the command:

```
ls /dev
```

It is prudent to remove device nodes for non-existent hardware. For example, on a system that has only a single floppy drive, device nodes for floppy drive one can be removed with the command:

```
rm /dev/rx1*
rm /dev/rrx1*
```

These commands remove both the standard and raw device nodes, double- and single-density, of floppy drive one. Notice that the floppy drive itself does not disappear.

1.3.3 Does VENIX Need Reconfiguring?

Most installations will require reconfiguration of VENIX before the system is fully functional. See the section "Reconfiguring VENIX" for details on doing this. You will need to reconfigure your VENIX system if one of the following is true:

- Your processor is not an LSI-11/23, 11/23 PLUS, or 11/24. While the distributed kernel will run on all processors, it is generated for an LSI-11/23 or 11/24, and will not be 100 percent reliable on anything other than these processors. In addition, the distributed kernel will not support split I/D space. To support the full features of your processor, you must remake a VENIX kernel specifically for your processor type, as described in the sub-section "Making VENIX" within "Reconfiguring VENIX".

- You have more than 256kb of primary memory. The distributed kernel has a 256kb limit. If you have more primary memory on your computer, and would like it available for user programs, then the kernel will need to be regenerated. Any kernel you regenerate will automatically use all memory available.
- You have devices different than those preconfigured. The devices your preconfigured kernel can handle are limited to the ones listed above. If you have different devices, or additional units or lines of the same devices, or non-standard addresses, you will have to regenerate VENIX to accommodate the differences.

Until you have regenerated VENIX, it is important to avoid accessing device nodes for hardware which does not exist, as this will cause VENIX to crash. Similarly, VENIX will also crash if you activate hardware VENIX has not been configured for.

1.3.4 Uses of VENIX File Systems

Most hard disk units are divided into two or more logical disk partitions. This is done both for convenience and for protection. The layout of your particular disk is described in the table in 'What Hardware is Accessible'. If the partition sizes are not to your liking, it is possible to change them by simply modifying a table in the device driver. This is described later in 'Adding and Changing Drivers.'

There are three uses for disk partitions:

1. The *system* partition (known as **/dev/*XXX*.sys**), holds the main VENIX commands, libraries, devices, and so on, as well as the 'swap area' for swapping processes. When VENIX is first booted, this is the only file system known. The system partition contains the kernel (**/venix**), as well as the system directories **/bin**, **/lib**, and **/etc**.

2. The *user* partitions **/dev/*XXX*.usr** (or **/dev/*XXX*.usr0**), **/dev/*XXX*.usr1**, and so on, are used to contain lesser-used VENIX commands and libraries, as well as user files. The first user partition is mounted under directory **/usr**, and is the only one that holds both VENIX and user files. The standard VENIX directories **/usr/bin**, **/usr/lib**, **/usr/dict**, and others are on

this partition. Additional user partitions are typically mounted under directories **/u0**, **/u1**, and so on, and are available exclusively for user files.

3. The *temporary* partition (known as **/dev/*XXX*.tmp** is used by compilers, editors, and other programs for temporary data storage. A clean file system is created on this partition and mounted under directory **/tmp** when VENIX goes from single-user to multi-user mode. Permanent files should never be left under directory **/tmp**.

1.3.5 Installing the USER AREA — Tape Distributions Only

This section should be skipped if your VENIX distribution was not on tape.

If you are installing VENIX from tape, then you have only restored about half of your system. The remainder of your system, which will be placed on the user area, is on the diskettes or tape marked USER AREA. If your system was delivered on hard disk or floppies, then the files are already there, and you should skip to the next section.

A file system should be created on the first user area. Consult the table in 'What Hardware is Accessible' to determine the name and size of that area. Type the command:

/etc/mkfs −b *name size*

where *name* is the name of the first user area, and *size* is its size in blocks. For example, to make a file system on the first user area of an RL02 disk, type:

```
/etc/mkfs −b /dev/rl0.usr 10240
```

Now that a file system has been made on the first user area, it should be mounted under directory **/usr**. The command to do this is:

mount *name* /usr

where *name* is the name of your first user partition. For example, on RL02 disk:

```
mount /dev/rl0.usr /usr
```

If your distribution was on tape, load the USER AREA tape on unit zero. Type the commands

```
cd /
tar xvf /dev/mt0 ./usr
```

The names of the files will be given as they are extracted from tape.

1.3.6 Initializing User File Systems

At this point, file systems exist on your system and first user areas. It is now time to create file systems on any other user areas that exist.

Examine the table in 'What Hardware is Accessible' to determine the name of the second and other user areas and their size in blocks. For the second and additional areas, type

/etc/mkfs −b *name size*

where *name* is the name of the area, and *size* is its size in blocks.

WARNING: do not create a file system on the first user area or on the system area, since these areas contain valuable files.

For example, on an RL02 disk:

```
/etc/mkfs −b /dev/rl1.usr 20480
```

Again — be careful to avoid overwriting the first user area.

The **−b** flag given in the above command causes a bad-block check to be made as the file system is created. Any bad blocks found are marked unusable and collected in a file called 'Bad.Blocks' which should never be removed or touched. This flag need not be used if the disk is error-free.

1.3.7 Automatically Mounting via /etc/rc

The command file **/etc/rc** is automatically executed every time VENIX goes from single-user to multi-user mode. **/etc/rc** already contains a number of commands in it to clear several tables, start up a few background processes, and initialize and mount the temporary partition on directory **/tmp**. Commands should be added to **/etc/rc** to mount all the user partitions in their appropriate directories.

The command to mount the first user partition **/usr** has been placed in **/etc/rc**, but preceded with a ':' so that it will not be executed. Remove the preceding ':' from the **mount** command so that the first user partition will automatically be mounted under directory **/usr**.

The other user partitions are normally mounted under directories **/u0**, **/u1**, and so on. Empty directories of these names are provided for this purpose. The directory names used are arbitrary, however, and you may create your own directories for mounting purposes if you wish. Commands should be added to **/etc/rc** to mount all the additional user partitions. For example, for an RL02-based system, **/etc/rc** would now contain the lines:

```
mount /dev/rl0.usr /usr
mount /dev/rl1.usr /u1
```

The directories you mount your disk in (**/u0**, ...) are empty before a disk partition has been mounted beneath them. Once the partition has been mounted, the files and directories there appear 'magically' in the specified directory. Don't copy files into **/u0**, **/u1** or other mounting directories before they are actually mounted under, because any files already there will become inaccessible once mounting is done.

The 'temporary' partition is not mounted when you are in single-user mode, as you are now, which means that the **/tmp** directory remains located on the system partition. Since free disk space may be tight on the system partition, it is best to avoid unnecessary compiling, editing, and other activities when you are still in single-user mode. These activities make use of temporary files which, placed under directory **/tmp**, are located on the system partition where free disk space may be limited.

1.3.8 List of File Systems in /etc/checklist

The file **/etc/checklist** should now be edited to contain an entry for each disk partition. **/etc/checklist** contains a list of file systems to automatically be checked whenever you run the **fsck** command without arguments. **/etc/checklist** should already contain a line looking something like

```
/dev/rrl0.sys:system:
```

The name at the beginning of the line (i.e., up to the first colon), is that of the system partition; the words following (up to the second colon) are an English-language comment. The device names given here have an extra 'r' in front of them to designate the 'raw' version, which can be **fsck**ed slightly faster than the ordinary block version. (CAUTION: on all RM type disks, the 'raw' version of the first user partition should not be used, since it is inverted. Use the standard 'block' version instead.)

For example, on an RL02-based system, **/etc/checklist** should contain the three lines:

```
/dev/rrl0.sys:system:
/dev/rrl0.usr:user area:
/dev/rrl1.usr:second user area:
```

1.3.9 Which Terminals Come Up Multi-User?

When you bring the system up from single- to multi-user, as described in the section 'Going Multi-User', various terminals will become active (give a 'login:'), depending upon the instructions in the table in the file **/etc/ttys**. Only terminals made 'active' in this way can be logged in to. They may be communicated with in either case. (See **init**(8) for a description of the initialization process.)

Examine the table in the section 'What Hardware is Accessible' for a description of the terminal lines for which VENIX has been preconfigured. (The console terminal (**/dev/console**) is standard for all PDP-11's and will always be correct.) If the register or interrupt vector address given for a line does not correspond with your hardware, then DO NOT attempt to use the lines until you have reconfigured VENIX. VENIX and the serial line hardware must be in agreement about these addresses before a line can be accessed—VENIX will crash if they are not. See the section 'Reconfiguring VENIX'.

The **/etc/ttys** file lists terminals by their **/dev** directory names, and gives a description of the terminal. The first character in an entry is '0' if the device will be inactive, or '1' if active. The second character is usually '2' to indicate a CRT, or '4' for a hardcopy terminal. (If the device is inactive, however, the second character is not important.) The rest of the line is the device name, and must refer to an entry in **/dev** corresponding to a valid line. For example,

```
12tty01
```

indicates that the terminal on line **/dev/tty01** is to become active, and is a CRT.

```
14tty02
```

indicates that **/dev/tty02** is to become active, and is a hardcopy terminal. See **ttys**(4) for more details.

If you are in doubt as to which device name corresponds to which terminal, try sending a little output to each one. For example:

```
echo "I am tty00" > /dev/tty00
echo "I am tty01" > /dev/tty01
        •
        •
        •
```

This will confirm that the terminals devices are correctly configured into the system, and tell you the name of each one. It is a good idea to run this test on every line before marking it 'active' in **/etc/ttys**.

Note that 'active' here means that the device can be used as an interactive port at which users can log in. Devices which are inactive can still be used to send or read data from; for example, a serial line which has a printer on it would never be set 'active' unless the printer had a keyboard on it and would be used interactively. If your console terminal is a hard-copy unit, you are free to declare it inactive and thus limit its uses in multi-user mode to logging system error messages (sent there automatically) or printing files.

You can specify as many active lines as you are licensed for (typically eight or sixteen); additional entries will be ignored.

1.3.10 Configuring the Line Printer

The 'special file' **/dev/lp** is a generic device entry referring to whatever device is used as the system printer. If the printer is running from a serial line, then **/dev/lp** should point to that line, as a duplicate of **/dev/tty??**. This can be accomplished by first identifying the **tty** line that the printer is on, and then 'linking' the **lp** entry to it. For example, if the printer were on line **/dev/tty02**, then the command would be

```
ln /dev/tty02 /dev/lp
```

If a 'real' line printer exists (LPV-11 or an electrostatic printer/plotter), then the line printer driver should be added into the system (see 'Adding Drivers' below). The entry **/dev/lp** should then be created with device numbers corresponding to the printer driver (see the section 'Making Device Nodes.')

Once the **/dev/lp** entry has been created, the spooler command **lpr**(1) can be used to queue files for printing on this device. The sources to this command are provided (see **/usr/src/lpr.c**), so that it may be recompiled with options appropriate for your hardware (number of lines per page, use of form feed codes, etc.).

Don't confuse the spooler command **lpr** (located in directory **/bin**) with the special file ("device node") **/dev/lp** or the LPV-11 device driver source **lp11.c**.

1.3.11 Completion of Initialization Exercises

If the system is not running from the previous exercise, or it was shut down and halted, then boot the system as described previously under the section 'Booting'. If you have mounted any user partitions, then unmount them with a command of the form:

```
umount /dev/rl0.usr
```

At this point the following things should have been done for your installation:

1. File systems should be initialized on all the user partitions you intend to use (see 'Initializing User File Systems').
2. **/etc/rc** should contain **mount** commands to mount user partitions under directories when multi-user mode is entered (see 'Automatically Mounting via **/etc/rc**').
3. **/etc/checklist** should contain lines to indicate which disk partitions you want checked (see 'List of File Systems in **/etc/checklist**').
4. **/etc/ttys** should contain lines to indicate the terminals which will become active in multi-user mode (see 'Which Terminals Come Up Multi-User'). All of the entries should refer to usable terminal lines.

1.4 GOING MULTI-USER

The initialization exercises you just completed need be done only once for your installation, or when you are reconfiguring your system. The following steps, however, should be done each time you bring your system up. If the system is not running after completing the previous exercises, or it was shut down and halted, then boot it again as described previously under 'Booting.'

1.4.1 Using fsck

All the file systems should be checked for consistency (especially after a crash) before they are mounted. The commands in **/etc/rc** will mount the file systems as soon as the system goes into multi-user mode, so the check should be done now, while the system is still in single-user mode.

The **fsck** command checks all the standard disk partitions you specified in **/etc/checklist**, and lists each partition as it checks it. Type

```
fsck
```

to perform this test. See **VENIX MAINTENANCE** for a more complete description.

1.4.2 Setting the Date

After running **fsck**, you can enter the current date by typing

> **date** *yymmddhhmm*

yy is the current year (e.g. 84), *mm* is the month (01 to 12), *dd* is the day (01 to 31), *hh* is the hour (00 to 24), and *mm* is the minute (00 to 60). If the date is not set, then the system uses the date and time it last remembers. If the year, month or day is omitted, the last remembered value is taken by default. For example,

> **date 8506250946**

sets the date to 9:46 in the morning, June 25, 1985. Now if you realized you were off by two minutes, simply typing

> **date 0948**

will set the date to 9:48 of the same day. The system tells you the date you set.

Incidentally, you don't have to be in single-user mode to set the date, but it is usually most convenient to do it then. If you decide to set it in multi-user mode, you must be super-user.

1.4.3 Entering Multi-User Mode

When the health of the file systems has been verified with **fsck** and the date has been set, the system may be brought to multi-user by typing a CTRL-D. (Note that typing **login** is not sufficient.)

The commands in **/etc/rc** are now automatically run, which should mount all the user partitions. Messages should appear on the console indicating that multi-user mode is being entered, and giving the current date. After a few seconds, a 'login:' message should appear on all terminals specified as active in **/etc/ttys**.

1.4.4 Logging in as Super User

You should now log in as the 'super user'. The super user has powers far beyond those of mortal men and women ... namely, the ability to access any and all files, as well as to execute many privileged commands associated with system growth and maintenance. The super user login name is 'root' which should be typed in after the 'login:' prompt on any of the terminals. The initial root password is 'gnomes' which should be typed immediately after the prompt 'Password:'. For security purposes, the password you type in never appears on the terminal, even though VENIX will read and understand it. If you mistype either the name 'root' or the password 'gnomes' you will receive another 'login:' prompt.

After you have correctly typed in the password, you will see the words:

```
Welcome to VENIX

Beware: you are the super user!
```

And underneath, a prompt 'SUPER>' will appear. The 'Beware ...' message is simply a warning that you are the super user. The 'SUPER>' prompt is a repeated reminder of your status. The super user password should be changed from 'gnomes' to something else if a more secure system is desired (see the section 'New Passwords').

1.4.5 Verifying the Disks and TTYs

If this is the first time that you have brought VENIX to multi-user mode, you should check that the disk partitions have been mounted and terminal lines made active. Make sure that 'login:' prompts have appeared on all the terminals specified as active in **/etc/ttys**. Type the command

```
mount
```

and verify that all the partitions you intended to mount with commands in **/etc/rc** have indeed been mounted. In particular, it is vital that the first user partition be mounted under **/usr**, since that is where system commands reside and where user accounts will be created. If things aren't right, double-check your typing of entries in the **/etc/rc** and **/etc/ttys** files. Otherwise, you can now begin adding other users to the system.

1.4.6 Crashing Going Multi-User

If VENIX crashes as the system goes to multi-user mode, there is probably a configuration problem with either some of the user disk partitions or the terminal lines. Reboot the system, and while still in single-user mode, type in all of the **mount** commands you placed in **/etc/rc**. Test out all the terminal lines by sending output to them as described in the section 'Which Terminals Come Up Multi-User'. You may need to avoid using some of the hardware until you have reconfigured the kernel.

1.5 NEW USERS

1.5.1 Creating New Users

There are three steps to creating new user accounts under VENIX: editing the password file, creating the home directory, and creating the user's login initialization file, **.profile** or **.login**. You must be logged in as the super-user 'root' to accomplish these tasks.

1.5.1.1 Editing the Password File

The system coordinator (as super user) can install new users by editing the password file **/etc/passwd**.

Each line in the file corresponds to a separate user, and contains an entry of the form:

name:password:UID:GID:empty:directory:shell

where

name is the user's name (lower-case letters only).

password
is the user's encrypted password. After the new user logs in, he will put in his own password, using the **passwd** command. Do not enter text into this field yourself. Since this field is encrypted, when set it appears as nonsense letters.

UID is the user's numerical ID number, which should be unique among all user ID's. Numbers 1-128 may be used; user ID 0 is reserved for the super user 'root'.

GID is the user's numerical group ID number, indicating the group he or she belongs to. Until you set up different user groups, you can use group ID number 10 for everyone. If you wish to divide your users into different groups, then the file **/etc/group** can be edited to create new groups, and users can be given group ID numbers assigning them to one or another group. Numbers 1-128 may be used; group ID 0 is reserved for the super group (see the "VENIX Maintenance" chapter for details).

empty is an unused field.

directory
is the user's 'home' directory, where he is placed when first logging in.

shell is the user's shell interpreter, which is run when he logs in. If this field is left blank, the standard 'Bourne' shell is used (see **sh**(1)). Users who wish to use the C shell **csh**(1) should put the pathname **/bin/csh** in this field; note that this field is not terminated with a colon. (While the Bourne and C shells are in many respects identical, the latter offers several unique features such as a 'history' mechanism for easy reissuing of past commands. See the two shell tutorials in the *User Guide* for details.)

The following commands could be used to enter a new user 'holmes' in the password file. The line editor **ed**(1) is used to edit the password file:

```
SUPER> ed /etc/passwd
188
$a
holmes::10:10::/usr/holmes:
.
w
217
q
```

This appends an entry for 'holmes' to the end of the password file, with user

ID 10, group ID 10, and login directory **/usr/holmes**.

1.5.1.2 Making Home Directories for New Users

The home directory given for the newly installed user must now be made, and the owner of the directory be changed to that user. The directory chosen above was **/usr/holmes**. Typically, all user directories are placed under directory **/usr**, although it is perfectly acceptable to place them on another mount~~ed file s~~ystem, for example **/u1**. But don't place user directories under a directory which is not mounted on some user partition.

After creating a user entry, login as that user to make sure the password file is correctly edited. For example, for user 'holmes' these commands would be given:

```
SUPER> mkdir /usr/holmes
SUPER> chown holmes /usr/holmes
SUPER> chgrp other /usr/holmes
SUPER> login holmes
$ l -a
total 2
drwxrwxr-x 2 holmes        64 Sep 23 12:07 .
drwxrwxrwx 7 root         208 Sep 23 12:04 ..
$ login root
Password: gnomes
SUPER>
```

In the example above, **/usr** refers to the top directory on a file system in the user area; the command **mkdir /usr/holmes** makes a directory called **holmes** there. The command **chown holmes /usr/holmes** changes the ownership of directory **/usr/holmes** to holmes, and **chgrp other /usr/holmes** changes the group ownership to 'other' (otherwise the group owner would be the root's group, 'system').

The new user entry is tested by logging in under the new name **(login holmes)** and listing all the files **(l -a)** (that's the letter 'l' not the number one). The only entries in this new directory should be '.' and '..', referring to the current directory (**/usr/holmes**) and its parent directory (**/usr**). Finally, a 'login root' and 'gnomes' will get the system coordinator back to super user status.

The following problems may arise when the new user tries to login:

- The user receives the message 'No directory'. This means that the shell could not find the login directory, as specified in the password file. Either that directory is not present, or the wrong number of fields are in the password file, causing the system to be confused about what field holds the directory name.
- The user receives a 'Password:' prompt, although the user has not designated a password. The user may have typed his or her name incorrectly and should try logging in again. Alternatively, the password file entry for that user might be jumbled.
- The user can not access his or her files. Either the ownership of the files was not correctly preset for the user, or the password file entry for that user has bad user or group ID numbers.

In any of these cases, refer back to the **passwd** command file to correct any mistakes in the file entry.

1.5.1.3 Setting Up .profile and .login Files

Now that the user account and directory is set up, a login initialization file should be made, either by the system coordinator (as super user) or by the new user. This file will be called **.profile** if the user is running the standard Bourne shell, or **.login** if the user is running the C-shell. (The choice of shells was made when the **/etc/passwd** file was edited, as shown previously.)

These files contain commands which are executed automatically whenever the user logs in. They are used to assign a custom prompt for the user, and to set 'environment' variables to indicate, for example, the type of terminal the user is on or the locations of personal command directories.

An initialization file is a convenience, not an absolute necessity. The commands it contains could always be typed by hand each time the user logs in. But this would be rather tiresome; we suggest you take advantage of the initialization file capability, and let the computer do the work for you!

We have left sample **.login** and **.profile** initialization files in the guest home directory, **/usr/guest**. You may copy one of these into the new user's home directory. We'll continue our example for new user **holmes**, assuming the he is running the standard Bourne shell (**sh**(1)), and will therefore require a **.profile** file rather than a **.login**. Corresponding details for the C-shell are given afterwards.

```
cp /usr/guest/.profile /usr/holmes
cd /usr/holmes
ed .profile
```

We are now in the **ed**(1) editor, ready to modify the file. (You may wish to use the screen editor **vi** or **ice**). The first thing to do is change the user prompt from 'GUEST:' to 'holmes:'. The line in question is

```
PS1="GUEST: "
```

To move to that line, we type

```
/GUEST/
```

and to change, and verify, the new prompt:

```
s/GUEST/holmes/p
```

The new line should now be printed.

A very important shell variable is **TERM**, which is used to specify the type of terminal being used. The complete list of terminals known to **vi** is given in **termcap**(5). For example, an entry for the DEC VT-52 should read

```
TERM=vt52
```

This variable is examined by the full screen editors such as **vi** (but not **ice**) and programs such as **more**(1). The section 'Setting Up Screen Editors' also shows a mechanism for automatically determining the terminal the user might be on.

Another shell variable we might wish to modify is **PATH**, which contains a list of directories, separated by colons, in which the shell looks to find commands. The standard locations to look in are the current directory (.) and the VENIX system 'binary' directories **/bin** and **/usr/bin**. (If you list the contents of these last two, you may see some familiar names.) Users frequently have their own private command directory called **bin** under their home directory. To add this name to the standard **PATH** for **holmes**, insert a new line at the beginning of the file with the editor commands:

```
$a
PATH=$PATH:$HOME/bin
.
```

The **PATH** variable is set to its original value plus the new directory **$HOME/bin** **$HOME** is a shell variable equivalent to the user's login (home) directory, in this case **/usr/holmes**. The expression **$HOME/bin** then corresponds to the directory name **/usr/holmes/bin**. (We haven't actually made such a directory yet, but when we do, the shell will search it to find any command name typed by the user.)

The **EXINIT** variable is useful for users of the **vi** and **ex** editors. It contains initialization commands which are read by these editors whenever they are invoked, and can be used to customize function keys and preset any of the numerous editor parameters. See the **vi** documentation for details.

If holmes were running the C-shell, the sample file **.login** should be copied over from **/usr/guest** to **/usr/holmes**. The following lines give the equivalent C shell versions of the variables initialized above:

```
set prompt="holmes: "
setenv TERM vt52
set path=(. /bin /usr/bin ~/bin)
```

The syntax used is somewhat different than that of the **.profile** file. The **setenv** ('set environment') command is used for variables like **TERM** which are to be passed to programs the user runs, not just used by the C shell itself.

The sample **.login** file also contains the entry

```
set history=20
```

This causes the C shell to remember the last twenty commands typed by the user, so that it can recall and re-issue them if the user requests. This is a terrifically useful feature; see the C shell documentation in the *User Guide* for more details.

Other commands can be inserted within **.profile** or **.login** files to print login messages, for example

```
echo "Glad to have you aboard, sir."
```

or perform routine login functions. A favorite is

```
/usr/games/fortune
```

which will print a few words of wisdom on each login. (WARNING: some of the 'fortunes' are rather odd, and should not be misinterpreted as system error messages.)

1.5.1.4 New Passwords

If system security is desired, then you should give yourself a password. Type in:

```
passwd
```

On the screen it will say, 'Changing password ...' and then VENIX will prompt:

```
New password:
```

Type in a password which has at least four characters. You will be prompted to type the password a second time. VENIX prompts:

```
Retype new password:
```

The password will not be visible on the screen (to keep the password confidential), but it has been entered. The next time the you log in, when the password prompt appears, type in the password.

The super-user can assign the password for another user with the command

passwd *user-name*

1.6 SHUTTING THE SYSTEM DOWN

If the system is multi-user, it should be first brought down to a single-user system by logging in as 'shutdown', i.e.

```
login shutdown
```

The password is **gnomes** as it is for the super-user. For a more secure system you may wish to change the password with the **passwd**(1) command. (You will probably wish to keep the root and shutdown passwords the same.)

shutdown verifies that all users (except super-users) have logged off and background processes have finished (except for system stuff). (If any non-super users are still logged in, you are told this and given a chance to warn them; **shutdown** asks you for the number of seconds you wish to wait before becoming single-user, and then warns all logged-in users at 30 second intervals.) Then all mounted file systems are unmounted and a small message about **sync**ing the system is printed on the console. In a short while, another 'SUPER>' prompt appears on the console device, indicating that the user is a super-user. The system is now single-user. You can play with the system in single-user mode if you like; a CTRL-D would again cause the system to go multi-user.

If you are booted onto a venix kernel named other than **/venix** (presumably for testing purposes), shutdown will not work. In this case, you must bring the system to single-user mode by hand, by first unmounting all file systems and then typing:

```
kill −1 1
```

The **kill** command signals the **init** process, (process ID #1) to switch to single-user mode. You must be the super-user to do this.

Before you turn the power off or hit the BOOT switch, you absolutely must run the **sync** command. This is required because VENIX does not, for efficiency reasons, necessarily physically write out all data to the disk when a program logically writes data. The **sync** forces all write-behind data to be flushed out to the disks.

After the **sync** the processor can safely be halted from the front panel and powered down. You may want to write protect the disks before powering down, although this should be unnecessary.

If you occasionally forget to type **sync**, or unplanned power downs occur, disks may be slightly corrupted. Again, the **fsck** command will check a file system on a disk, and if required, interactively repair it.

1.7 BACKING UP VENIX

It is very important that duplicate copies of VENIX be maintained, in case your running system is damaged and becomes unusable, or system files are lost. First of all, an exact duplicate of your original VENIX distribution should be made once, since the original copy may become damaged. Secondly, backups of any customizations you have made to VENIX should be done every so often, so that they can be easily reapplied should you need to revert to the original copy.

According to the terms of your VENIX license, you may create duplicates of VENIX for backup purposes only. ALL DUPLICATE COPIES OF VENIX MUST BEAR A COPYRIGHT NOTICE AND SERIAL NUMBER IDENTICAL TO THAT OF THE ORIGINAL.

1.7.1 Backing Up the Original Distribution

The following steps will create a copy of your original distribution media that can be loaded from scratch, just as your first installed VENIX.

1.7.1.1 Backing Up the Distribution on Floppies

If your distribution was on floppy diskettes, use the following procedure for each of the VENIX diskettes: place the original diskette in drive zero, and a blank diskette in drive one. If you have RX02 diskettes (988 blocks/diskette), use the command:

```
dd if=/dev/rrx0 of=/dev/rrx1 bs=4b count=247
```

If you have RX50 diskettes (790 blocks/diskette), type:

```
dd if=/dev/rf0 of=/dev/rf1 bs=10b count=79
```

1.7.1.2 Backing Up the Distribution on RL02 Disk

If your distribution was on RL02 disk, place a blank error-free disk in drive one, and type the following commands:

```
sync
dd if=/dev/rrl0.all of=/dev/rrl1.usr bs=50b count=409
```

(The **sync** is given first to flush system buffers to disk.) The complete RL02 disk in drive zero will be copied to drive one. This copy will of course contain any modifications you have made to VENIX so far.

If you are not sure whether the spare pack is error-free or not, you can check it with a **mkfs** command before copying the image, for example:

```
/etc/mkfs -b /dev/rrl1.usr 20480
```

If any errors at all are reported, the disk should not be used for a system image back-up. (Unless fatal errors are reported, however, the disk could still be used as a mountable user file system.)

1.7.1.3 Backing Up the Distribution on Magtape

If your distribution was on magtape, change to a directory mounted on a disk partition that has at least 2mb or 4000 blocks of free space Use the **df**(1) command to ascertain the number of free blocks on each partition. (Neither the system or the temporary partition will have this much free space.) For example, if your first user area has 4000 free blocks, type:

```
cd /usr
```

For each of the two tapes, do the following. Mount the tape in drive zero, and copy the tape image to disk with the commands

```
echo > tempfile
```

(to create an empty file) and then

```
dd if=/dev/mt0 of=tempfile
```

Now mount a blank tape in drive zero, and copy the image to it with:

```
dd if=tempfile of=/dev/mt0
```

When you have done this for both tapes, remove the image file:

```
rm tempfile
```

1.7.2 Backing Up Your Customizations

You should also create a record of all modifications made to your original system. The **tar** utility has a **d** option which causes it to operate only on files modified within a given number of days. This is useful for backing up only those file you have modified. Determine the number of days since you loaded your VENIX system, and see the description of **tar** in **VENIX MAINTENANCE.** (WARNING: any file which was only edited before you set VENIX's date may have an old modification time. Files such as **/etc/rc** and **/etc/checklist** may be in this category. Use the **touch**(1) command to change the modification dates of these files to the present.)

1.8 BACKING UP USER FILES

The importance of backing up user files cannot be over-emphasized! Any computer system will crash from time to time, due to anything from power-fails, hardware problems, software bugs, to the phase of the moon. Most often this will happen without the loss of any significant data. However, there will be occurrences where significant loss occurs. Another hazard is accidental removal of files by users, often due to erroneous wild card filename specification.

Backing up is a task which is widely neglected, and often the cause of tremendous anxiety and expense. Unless you wish to claim another entry in the annals of computer catastrophe, you will back up your user files thoroughly.

There are a number of possible ways to back up user files, but the best tool for almost all purposes is the **tar** command. **tar** allows you to save and restore individual files or directories, and can back up on any media, including floppy, tape, and hard disk. Among the other virtues of **tar** is that it is a *de facto* standard among UNIX-derived systems, so that files saved in **tar** format are likely to be readable by any other UNIX system (and of course any other VENIX system) that can handle the same media type. Use of **tar** is detailed in **VENIX MAINTENANCE**, so it will not be discussed here. **tar** has an entry in the *REFERENCE MANUAL* as well.

1.9 RECONFIGURING VENIX

The files needed to remake the VENIX kernel exist under the **/usr/sys** directory. The files and directories there are:

/usr/sys/**conf**
: The 'configuration' directory, containing tables of information describing the linkage to the drivers, both from the VENIX kernel and the hardware. The command

```
make venix
```

from within this directory will regenerate a VENIX kernel. Key files in this directory are: **config**, the master configuration file; **c.c**, the C-language branching tables; and **l.s**, the assembly interrupt vector interface.

/usr/sys/**dev**
: The device driver directory. All the sources to VENIX device drivers are given here, for example **dl.c**, the DL-11 driver, and **rl.c**, the VENIX RL01/02 driver. Documentation for all drivers can be found in section seven of this manual.

/usr/sys/**SYSLIB**
: This is the system library, in 'archive' (**ar**(1)) format, for all non-split I/D processors. It contains the object modules to the kernel itself, and need never be modified by the user.

/usr/sys/**SYSLIB.ID**
: This is the system library for split I/D processors.

/usr/sys/**DEVLIB**
: This is the device driver library. Drivers are compiled and then added to this library with the **ar** command.

The directory **/usr/include/sys** contains '#include' files used by drivers and the kernel. The only file subject to modification by users is **/usr/include/sys/param.h**, which defines system parameters that may be customized for a particular installation.

The steps needed to define and regenerate a new VENIX system are described in the following sections. They are:

1. Editing **config** to include references to all devices on the system.

2. Editing options with the driver sources; compiling them; and adding them to **DEVLIB**.

3. Linking the kernel, by running '**make venix**'.

4. Creating new device nodes in directory using the **mknod** command in **/dev**.

1.9.1 Editing Config

The **config** file, in **/usr/sys/conf**, contains an entry for each device on the system. Each entry is in the form of a C-language '#define' statement. The possibilities are given in comments, and if selected should be typed in exactly as given. For example,

```
#define TM
#define DA
```

would be used to indicate magtape and D/A devices. Delete all preset entries in this file referring to devices which are not in your computer.

This file also has an entry called **SYS_DISK**, describing the type of hard disk VENIX will run from, that is, where the root file system and swap area will be. The possible names are shown in comments; for example,

```
#define SYS_DISK SRL
```

indicates that an RL01 or RL02 disk will be used.

The entries in **config** can be used to describe most combinations of hardware. However, there are limitations (described within the file): for example, **config** can only define one type of RK disk (either RK05 or RK06/07). In cases where **config** is insufficient, or when a customer-written driver is being added to the system, then the file **c.c** must be edited to correctly link in drivers. This is described in 'Modifying c.c'. Normally, however, the user need not tamper with **c.c.**

1.9.2 Adding and Changing Drivers

1.9.2.1 General Info

The sources for the supplied drivers are in **/usr/sys/dev**, and are all in the C language. Generally, each hardware controller has a separate driver. Documentation on the user interface to the driver may be found in section seven of this volume. The procedure for adding a new driver to the system is:

1. Examine the options within the driver (i.e. register address, number of units) and set those appropriate to your hardware.
2. Compile the driver and add it to the device library **/usr/sys/DEVLIB.**

If you need to create a new driver for some device, probably the fastest method is to take the closest existing driver and make modifications to it, preferably in stages, where each stage can be tested before proceeding to the next. The general approach to drivers and what they need to do is covered in chapter five in the section "The VENIX I/O System". That document and the source listings which serve as examples are the primary tools available to help in writing a driver.

The debugger **adb**(1) can be used to examine the running VENIX kernel and the driver being debugged **(adb /venix /dev/kmem)**, and can show you the values in driver variables (see "Driver Variables").

1.9.2.2 Changing Device Register Addresses

The addresses of device registers are given by constant values at the beginning of the driver. They are preset to standard locations, and must be set to match the addresses your hardware is actually set to. For example, the DZ driver contains the line

```
#define DZADDR  0160100                 /* Base address */
```

defining the base address of the DZ registers. These numbers can be changed as needed.

The DL-11 driver is unusual in that is has a base register address for each line. Most other drivers have only a single base address, no matter how many lines or units are present.

1.9.2.3 Modifying Disk Partitions

Drivers for most hard disks have a *partition table* which describes the size and location of each logical disk partition. The original ordering is given in the write-up of that particular device (see the table in "What Hardware is Accessible?". It is possible to change these partitions if they do not suit you, or extend the table for additional disk units.

The exact format of the partition table varies slightly between drivers. The table is in the form of an array, each row having two numbers in it. One indicates the size of the partition, and the other, the offset of the partition on the disk. The partition size is usually in blocks (one block = 512 bytes); the offset is measured in cylinders (some block multiple) for larger disks, and measured in blocks for smaller ones. The choice of units should be clear from the driver source. The size of the array itself, that is, the number of partitions, is based on some constant given as the array dimension.

The following table is taken from the RL01/02 disk driver. The RL01 and RL02 disks use the same driver; depending on which disk is actually being used, the partition table is changed for 5mb or 10mb disks. The following table is for 2 RL02-sized disks, of length 10mb (20480 blocks).

```
struct {                            /* disk partitions (4 per physical) */
        unsigned nblock;            /* number of 512 byte blocks */
        unsigned oblock;            /* offset in blocks for first block */
} rl_sizes[4*NRL] {
        20480,  0,                  /* drv 0: 10mb  complete disk       */
        7840,   0,                  /*                 4.0mb root/swap */
        2400,   7840,               /*                 1.0mb tmp & pipe*/
        10240,  10240,              /*                 5.0mb USER      */
        20480,  0,                  /* drv 1: 10mb  complete disk       */
        10240,  0,                  /*                 5.0mb USER      */
        10240,  10240,              /*                 5.0mb  USER     */
        0,      0,                  /*                 unused partition */
};
```

The number of partitions is given by the dimension '4*NRL'; NRL is the number of RL units, and for each RL unit 4 partitions are given. The minor device number of each partition is given by its location in the array, starting with zero. The minor device number for any partition is therefore the index of that partition in the partition table. As the above table shows, RL minor device number 0 refers to a partition beginning at block 0 on the disk, of length 20480 blocks. This partition covers the entire disk, and overlaps the next three partitions (7840 for the system area, 2400 for the temporary area, and 10240 for the user area.) Since this complete disk partition conflicts with the next three, it is usually unused.

The next three entries correspond to RL02 unit one. The first entry on this disk covers the entire disk; the next two cover half each. A single file system could be made on the first entry of 20480 blocks in length, or two 10240 file systems could be made on the second and third entry. Finally, the fourth entry is a place-holder only.

It is perfectly valid for partitions to overlap one another, as long as overlapping file systems are not created. In the case of RL unit 1, one must chose to have either two file systems of 10240 blocks each on partitions five and six, or one file system of 20480 blocks on partition four.

The offsets and sizes of partitions may be modified by the user to change the disk layout. However, the number of partitions for each disk unit (four, in the case of the RL) should not be casually tampered with, since that constant is used in several places throughout the drivers.

If a partition size is changed, a file system must be remade on it of the new length. A file system may be of shorter length that the partition (this is occasionally useful) but never longer. Changing the length of the system partition is a rather involved process, and not recommended for that reason.

1.9.2.4 Changing Disk Driver Options

Disk drivers have several options which can be invoked by uncommenting the appropriate '#define' statement in the source for the driver.

OLSEEKS

Most disk controllers have the ability to overlap seeks on a multi-drive system (except for the floppy controller). If you have more than one physical drive, then OLSEEKS should be defined.

ONEPHYS

Many emulating controllers will split up one large physical disk, say a winchester, into several logical DEC equivalent disks. In this case, ONEPHYS should be defined to avoid defeating the cylinder sorting algorithm as well as overlapped seeks.

SOFTECC

The DEC controllers for large disks (RK06 and up) require that the driver apply the error correcting code fix to the data, while some emulating controllers apply the correction internally. Define SOFTECC if the driver must apply the correction.

IMPSEEK

The DEC RL01/02 controller requires an explicit seek to the desired cylinder before a transfer request can take place. Many emulating controllers do not require this explicit seek, since an implied seek is done when the transfer request takes place. However in some cases, this feature must be enabled by a special jumper on the controller board. Define IMPSEEK if your hardware has implied seeks.

INVERT

On large disks with several large partitions (for example the RM02/03), it is generally more efficient to have the first partition inverted so as to put the i-node area next to the i-node area of the following partition. See section **RM02(7)** for a more complete discussion. Define INVERT for this feature. 'Raw' versions of inverted partitions should never be used.

EXTADD

Disk (or tape) devices with 22-bit addressing capabilities (most newer Q-bus or massbus devices), must have this option enabled in order to handle the full address space correctly, if the computer has more than 256kb of memory. If this option is not available in either the hardware or the driver, then the device can not be used as the system disk, and can not be accessed in raw mode in any case. DO NOT define this option if the controller does not have 22-bit capabilities.

1.9.2.5 Supporting Multiple Controllers

Most drivers handle only a single controller (see driver sources for exceptions). If you have two or more of the same type controller in your backplane, you will often have to do some driver modification to support multiple boards.

The simplest way to support a second controller is by simply duplicating the original driver and treating the two controllers as completely separate devices. The duplicate driver must be edited so that external references do not conflict with those of the original driver; this means that all references to symbols like 'ddtab,' 'ddopen,' 'ddclose,' should be changed to something like 'dd1tab,' 'dd1open,' 'dd1close,' and so on. Be careful that in modifying the external names, you don't create conflicts within the driver by extending the names past the 7 characters of significance. For example, 'devaclear' and 'devaclose' are fine (unique to 7 characters), but 'dev1aclear' and 'dev1aclose' are not.

The two controllers must of course have separate register addresses and interrupt vectors, and these must be reflected in each driver as well as the **l.s** file. The **c.c** configuration file must also be edited to contain entries for both drivers. See the section "Extended VENIX Modifications" for details on editing **l.s** and **c.c**.

This approach can be easily extended to support an arbitrary number of the same controller. The penalty for this simplicity, however, is that each copy of the driver will duplicate much of the same code and consume additional kernel memory.

A more efficient technique is to create a single driver that can handle multiple controllers, with a range of minor device numbers dedicated to each one. Those driver routines which are passed a device number can use it to determine which board is being selected. A common interrupt routine can be used for multiple controller interrupts, and passed as an argument to similarly indicate the board causing the interrupt. See the section on modifying **l.s** for details.

1.9.3 Recompiling Drivers

Once a driver is edited and ready to be tried, it should be compiled and added/replaced into the driver library. Type

```
cc −c −O driver.c
ar r ../DEVLIB driver.o
```

The first command compiles the driver (the **−c** flag is used to suppress linking, and the **−O** indicates optimization). The second command replaces the driver into the device library. To see what drivers are in the library, type

```
ar tv ../DEVLIB
```

Extraneous drivers should be deleted by using **ar** with the 'd' option.

1.9.4 Making VENIX

If you have an unusual system configuration (for instance, non-standard interrupt vectors, or more than one RK05 controller), you will have to read the later documentation on customizing the files **c.c** and **l.s**.

If this is not the case, however, you can now create a new VENIX kernel. At this point, verify that all your drivers should have been compiled and placed in **DEVLIB**, and that the **config** file has been correctly edited. Move into the **/usr/sys/conf** directory and issue the command

```
make venixXX
```

where XX is the processor type of your machine. (Type simply

```
make
```

to get a list of possibilities.) This will cause the appropriate tables to be compiled and assembled, and, if there are no errors, a new VENIX kernel linked together. If there are errors, you will have to correct them and try a new **make**.

A frequent kind of error is an 'undefined *XXX*' which means you referenced some symbol (i.e. driver routine) *XXX* in the files **config**, **c.c**, or **l.s**, which couldn't be found. This is usually because a driver wasn't added to **DEVLIB**. 'Undefined' references to the symbols **end** and **edata** are spurious, and are produced when other symbols are not found; these messages will disappear when the other references are fixed. All C-language symbols will be preceded with an underscore.

If a split I/D kernel is being generated, 'undefined' errors will be given as 'relocation' errors, and the name of the routine will not be given. To obtain more precise error information, try remaking a non-split I/D version of the kernel, for example by typing

```
make venix23
```

Another common error message for non-split I/D kernels is:

```
venix too big: code + data = XXXXX
Total kernel size must be less than 49152 bytes.
Remove a driver or change buffer sizes in param.h
```

This is given when the size of the kernel, including all the drivers and buffer space, exceeds 48kb. The solution, as suggested in the message, is to remove one of the drivers or reduce the buffer sizes given in the file **/usr/include/sys/param.h**. The latter option is usually taken; the parameter **NBUF** should be adjusted in the **param.h** file. This parameter controls the number of disk buffer blocks, each one using up 512 bytes. It should be lowered only by enough to make the kernel fit in the 49152 byte space. Reducing NBUF by a little will only causes a minor performance penalty in VENIX; if it is brought too low (for example, close to the total number of simultaneously mounted file systems), the system performance will suffer seriously, and VENIX may hang up.

1.9.5 Making Device Nodes

If you have added a new driver to VENIX, or added new lines or units to an existing driver, you will have to create the "special" files for the device in the **/dev** directory with the **mknod**(1m) command. The command is of the form:

```
/etc/mknod /dev/name [b] [c] major minor
```

Either **b** or **c** must be chosen, depending upon whether the entry is for a **b**lock or **c**haracter device. (Disk drivers have separate entries for both.) **major** is the device's major device number, which is equal to its index number in the appropriate **c.c** table; **minor** is the device's minor device number. For disks, this is equal to the index number of the particular partition in the driver's partition table; for serial interfaces, this number indicates an entry into a table of port addresses. For other types of devices, the minor device numbers have different meanings (see the appropriate writeup in section seven).

If you list your **/dev** directory with the **l** command, you will see the major and minor numbers for each existing entry, separated by commas, just before the date.

For example, to create a special file for the user partition on an RL disk:

```
/etc/mknod /dev/rl0.usr b 2 3
```

and for the raw version:

```
/etc/mknod /dev/rrl0.usr c 13 3
```

The RL index (major device number) is **2** in the block table (for the mountable (block) version of the partition), and **13** in the character table (for the raw (character) version). The minor device number for this particular partition is always 3 (see **RL**(4)).

To create entries for 4 DZ lines:

```
/etc/mknod /dev/tty00 c 2 0
/etc/mknod /dev/tty01 c 2 1
/etc/mknod /dev/tty02 c 2 2
/etc/mknod /dev/tty03 c 2 3
```

1.9.6 Testing a New Kernel

You should test the new kernel before removing the current VENIX kernel. The following procedure is recommended.

Move the kernel you just created from **/usr/sys/conf/venix** to **/venix.test**. Shut the system down, and reboot on 'venix.test'. Instead of 'venix', type **venix.test** after the '&' prompt.

If the system does not come up because of some error in this newly created venix, you can boot again on 'venix' to get a running system. After you have successfully booted and tested **venix.test,** you can move **/venix** to **/venix.old** and **venix.test** to **/venix,** and from that point on boot normally on **venix.**

Keeping the previous **/venix** around in **/venix.old** accomplishes two things: if later on bugs are discovered, then the old version can be booted and checked to see if the bugs are a recent addition or have previously existed. Also, if for some reason the file **/venix** is corrupted, for example on a disk crash, then the independent **/venix.old** file can be booted unless the disk is seriously ill. See "VENIX Maintenance" on system crashes.

New drivers should be tested out while the system is in single-user mode. If a new disk driver has been added, file systems should be made and **fscked** on all the disk partitions. Check out both the raw and block versions for each partition (except inverted partitions, which should not have "raw" entries.) Output should be sent to all new terminal lines to test them out before they are made active login ports. Don't mount a disk or activate a login terminal line before this type of testing has been done.

The commands **ds** and **shutdown,** etc. which assume that the currently running system is named **venix,** will not work correctly if the current system is named something else. **ds** has a **−k** option which must be used to specify a kernel other than **/venix.**

1.9.7 Device Errors

If your device drivers are configured incorrectly or have bugs, a variety of errors may occur. The messages that accompany these are given at the end of "VENIX Maintenance." The following is a quick list of possible problems and fixes:

1. Check that the register address in the driver source and the interrupt vector addresses in **l.s** matches those set in hardware. This requires examining board jumpers, and checking register addresses in memory through the console switches or ODT. (See "Modifying l.s" for instructions on changing VENIX's vector addresses.)
2. Check that your /**dev** device nodes are numbered correctly.
3. Verify that the device controller is located in the correct backplane slot, as indicated by your hardware manuals. Check for bus interrupt grant and DMA grant continuity.
4. Place calls to the routine **printf** at strategic places in your driver. These act just as the normal user program **printf**s do, but cause information to be printed immediately on the console terminal. This may be useful for determining if a driver routine is being called, or for determining what values are being passed.
5. Use **adb** to examine kernel memory.

   ```
   adb /venix.test /dev/kmem
   ```

   starts you up. (Ignore the message 'not a core file.') From there, commands like

   ```
   0177560/o
   ```

   can be used (that command for example prints the value of your console DL register in octal). See **adb**(1) and "A Tutorial Introduction to ADB" in the *Programming Guide* for instructions on using the debugger. (Symbolic addressing will not work on split I/D kernels.)

1.9.8 Extended VENIX Modifications

The following passages describe more detailed modification to system tables and entries. These are necessary if your hardware configuration cannot be adequately described by the **config** file, or if you wish to fine-tune your system. Once you have made these modifications, go back to the section "Making VENIX" and remake your VENIX kernel.

1.9.8.1 Modifying c.c

The **config** file can be used to describe most system configurations; it is automatically included by **c.c** to prepare the appropriate driver entries. However, there are situations which can not be described correctly just through **config**, in which case you will have to modify **c.c** by hand— for example if you have both an RK05 and an RK06 drive on your system, or two different RL controllers.

Entries in **c.c** are parameterized with "#ifdef" and "#else" statements, causing conditional compilation of entries if the given preprocessor constant has or has not been defined. (After editing **c.c** and running

```
make venix
```

you can peruse the file **c.i**, the preprocessed version of **c.c**, in order to check that your entries were processed correctly. Ignore the blank lines and extraneous structure definitions.)

There are two tables which you may need to modify: **bdevsw** and **cdevsw**. The first contains entries for *block* devices, such as disk devices and magtape; the second contains entries for *character* devices, such as serial or parallel I/O, A/D's, as well as the raw versions of most block devices. Before an entry is used in the block or character table, it must be declared with an *extern* statement.

After the two table declarations, there is a set of numbers which indicate which disk partitions VENIX will run from. These should only be modified if you need to move your VENIX system to an entire different unit. See below under the section "System Partition Assignment".

Block Device Table

Each line in the **bdevsw** is of the form

```
&devopen,     &devclose,     &devstrategy,  &devtab,
```

The **dev** portion of each name should be replaced by the appropriate device type. For example, the entry for a magtape device is

```
&tmopen,      &tmclose,      &tmstrategy,   &tmtab,
```

The exact prefix can be found in the device driver source, and usually consists of the two or three first letters of the device name. Disk devices have no need for open or close routines, and use the special **&nulldev** entry in their place. For example, the entry for an RX floppy disk would be

```
&nulldev,     &nulldev,      &rxstrategy,   &rxtab,
```

Entries corresponding to non-existent devices are

```
&nodev,       &nodev,        &nodev,        0,
```

These are merely place holders. The difference between 'nulldev' and 'nodev' is that the former indicates that no routine is necessary to perform the function, while the latter indicates that the function should not be attempted, and that an error should be produced if it is called.

Character Device Table

The **cdevsw** table contains entries of the form:

```
&devopen,  &devclose,  &devread,  &devwrite,  &devioctl,
```

corresponding respectively to the routines to open the device, close the device, read from the device, write to the device, and perform I/O control functions on the device.

For example, the entry for a DL device is

&dlopen, &dlclose, &dlread, &dlwrite, &dlioctl,

The entry for the raw version of the RX disk device is

&nulldev, &nulldev, &rxread, &rxwrite, &nodev,

Note that the open and close entries are 'nulldev', indicating that no routine need be closed to perform those functions, and that the I/O control entry is a 'nodev' indicating that there is no I/O control for this device. Most disk devices do not have I/O control functions.

The following two special entries will always exist somewhere in the **cdevsw** table, and should not be removed:

&ttyopen, &nulldev, &ttyread, &ttywrite, &ttyioctl,

&nulldev, &nulldev, &mmread, &mmwrite, &nulldev,

The first corresponds to the 'tty' device (**/dev/tty**), a special device which always accesses the terminal of the program using it; the second accesses memory, and is used for debugging purposes.

System Partition Assignment

There are three basic partitions which the system knows about: **pipedev**, **rootdev**, and **swapdev**. **pipedev** is on the temporary partition, and is used internally by VENIX for buffering pipes. **rootdev** is the partition which the system itself runs on, holding the entire root file system, and **swapdev** is the partition used for holding swapped processes. **rootdev** and **swapdev** are usually on the same partition, and thus have identical values.

The names PIPE, ROOT, and SWAP are set in the **config** file to specify the partitions. The declarations are then used in the **c.c** file to initialize the variables **pipedev**, **rootdev**, and **swapdev**.

The **config** file declarations are of the form

```
#define PIPE        = ( (x1<<8) | y1 );
#define ROOT        = ( (x2<<8) | y2 );
#define SWAP        = ( (x3<<8) | y3 );
```

The **x** value in each set indicates the *major* device number of that partition (usually the same for all three). The **y** value indicates the *minor* device number, which is the index of that partition in the driver's partition table.

To prevent the swap area from overwriting the root area, the swap area is offset onto the partition to a location following the root file system. The **config** file variables and "SWAP__OFF" "SWAP__NO" are used in **c.c** to define **swplo** and **nswap**, respectively. The values

```
#define        SWAP__OFF 6816
#define        SWAP__NO  1024
```

are used for most disks. **SWAP__OFF (swplo)** indicates the offset (in blocks) of the swap area on the given partition. Since the root file system and the swap area usually reside on the same partition, **swplo** defines the length of the root file system area. **SWAP__NO (nswap)** is the size of the swap area. The sum of these two numbers should equal the length of that partition.

The minimal swap area should be around 100 blocks, for systems which will only be used in single-user mode and will not have to run background programs. Standard systems have a swap area of around 1000 blocks. Large systems often will have 2000 to perhaps 3000 blocks.

In very special cases, the swap area may be put on a separate device or partition which the system can access more quickly or has a faster transfer rate.

1.9.8.2 Modifying l.s

If your device has standard interrupt vectors (or none at all), then the entries you make in the **config** file should do everything for you. Otherwise, you will have to create and manually edit the **l.s** file. This is an assembly-language source file, which, like the **c.c** file, is parameterized with *#ifdef* and *#else* statements to reflect definitions in **config**. After running

[illegible]: 0 BOOT + SWAP
1 TMP + PIPE
2 ROOT
3
| USER
15

```
make venix
```

the preprocessed version of **l.s** can be see in **l.a.**

Everything to the right of a '/' indicates a comment. In the first half of **l.s** are entries of the form:

```
. = nnn^.
        ddio; brp+X.    / this is a comment
```

nnn is the interrupt vector in octal. *dd* is a two-letter abbreviation for the device (e.g. 'dr'), and *p* is the priority the device has.

The *X* is a optional decimal value ranging from 0 to 31 which is passed along as argument to the C-language interrupt routine. This is useful in drivers such as the DL driver that have a single interrupt routine which may be called through a number of different interrupts. A unique value *X* can be used at each interrupt, so the routine can determine exactly which vector was used. Most drivers do not use this feature, in which case the value of *X* should be left at 0.

For example, the entry for an RL disk is

```
. = 160^.
        rlio; br5+0.            / RL
```

These interrupt vector numbers must appear in order of increasing value in the file, and the vectors must be at least four bytes apart. This entry indicates that when the particular device interrupts, the routine **rlio** should be called, and that the priority should be set to 5. **rlio** is defined below.

In the second half of the **l.s** file are entries of the form

```
.globl  __ddintr
ddio:   jsr     r0,call; __ddintr
```

where **dd** is the name of the disk device. This is called by an interrupt at the previously defined location, and causes a subroutine call to **dd**intr, which is the interrupt service routine in the particular device driver. For example, the appropriate declaration for the RL driver would be:

```
    .globl  __rlintr
    rlio:   jsr     r0,call; __rlintr
```

1.9.8.3 Modifying param.h

The include file **/usr/include/sys/param.h** contains a number of parameters which may be changed in order to customize VENIX to your installation. In general these will not need to be adjusted; in any case, you should not tamper with them unless you have some in-depth knowledge of the system. Some of the more commonly modified parameters are:

NBUF	Number of disk buffers. Reduced sometimes if the kernel is too large after a 'make venix' command.
NMOUNT	Maximum number of mounted file systems. The 'root' system counts as one, too. If this limit is reached, the **mount** command will fail on attempts to mount additional file systems.
HZ	Ticks/second of the clock. Normally 60; set to 50 in Japan and Europe.
TIMEZONE	The time difference between local and GMT in minutes. For EST this is 5*60; for PST it would be 8*60. This parameter changes the timezone abbreviation given by the **date** command.
DSTFLAG	This flag, when set to 1, causes the time to shift to Daylight Savings automatically between the last Sundays in April and October.
NPROC	Maximum number of processes.

MAXUPRC	Maximum number of processes per user (normally 10).
RMEMB	A segment of memory may be reserved from VENIX, in which case it will not be touched by the operating system or counted as 'available' when VENIX is booted. This parameter defines the base of the reserved memory area, in units of 64 bytes. This is used sometimes if VENIX finds and tries to use memory between 248 and 256kb on 18-bit systems which map this to the I/O page.
RMEML	The length of the reserved memory area whose base is defined by RMEMB. Also in 64 byte units. If zero, no memory is reserved.

Again, the VENIX kernel must be remade after these are changed. Note that some of these variables have two entries: one for non-split I/D processors, and one for split I/D processors. Edit the instance appropriate to your system.

1.10 MISCELLANEA

1.10.1 Various Distributed Sources

Source code for several utilities is distributed with VENIX. Among these utilities are: the screen editor **ice** and the line printer spooler **lpr**. These sources may be found in /**usr/src**. Other 'goodies' may be included.

1.10.2 Setting Up Screen Editors

The simple screen editor **ice** is set up to work on VT52 or VT100 terminals or their equivalents. No special setup work is required to use it; see **ice**(1) for usage details.

The screen editor **vi**, on the other hand, can work on a variety of different terminals. For **vi**, look in the file /**etc/termcap** for a listing of the terminals which are supported. If your video terminal is not listed, substantial modification may be necessary. Consult your VENIX distributor. The file **/etc/termcap** should be edited to remove entries for terminals not used in your installation, as searching the entire large file will cause **vi** to be slow in starting up. See **termcap**(5) for details on the file format.

The section "Setting Up .profile and .login Files" discussed the setting of the **TERM** variable in the user's shell, used to tell **vi** and some other screen editors what type of terminal the user is on. The example in that section showed the **TERM** variable set to a single particular name. However, if users are logging in on several different terminals, it is convenient to have the shell set **TERM** to the correct name. This can be done with some simple shell programming in the **.profile** or **.login** files. The trick is to have the shell first run the **tty**(1) command to determine which line you are on, and then set **TERM** with the appropriate name. To accomplish this, the following statements may be placed in the **.profile** for users of the Bourne shell:

```
case `tty` in
        /dev/console)    TERM=vt52 ;;
        /dev/tty00)      TERM=vt100 ;;
        /dev/tty01)      TERM= .... ;;
        .
        .
esac
export TERM
```

The corresponding entry in **.login** for users of the C-shell is:

```
switch (`tty`)
        case /dev/console:
                setenv TERM pro
                breaksw
        case /dev/tty00:
                setenv TERM vt100
                breaksw
        case /dev/tty01:
                setenv TERM ....
        .
        .
endsw
```

Note the grave accents around the **tty** command. See the Bourne and C-shell documentation for details on these constructions.

1.10.3 Setting Up Nroff

nroff(1) formats text for different printing terminals. It knows about a number of different printers; however, if yours is not among them, you will have to write a new terminal descriptor table. See "NROFF Terminal Descriptor Table Format" in the *Document Processing Guide* for instructions.

1.10.4 Operating Without Floating Point Hardware

If your processor does not have the floating point option, then the **−f** switch on the C compiler must be used when compiling any programs that use floating point. The **−f** causes the floating-point simulator to be loaded in with the program. (The system, however, will not use the FP simulator unless the hardware is really missing.) Programs which heavily use floating point will, of course, run slowly if using the simulator instead of the real hardware. Attempts to execute floating-point code without the benefit of either hardware or simulator will result in 'illegal instruction' errors.

The programs **f77, pc, lint,** and **maze** will not work if floating-point hardware is not present.

Chapter 2

VENIX MAINTENANCE

2.1 INTRODUCTION

This document is a guide to administering a VENIX system. Topics covered include: maintaining file systems and managing disk space; backing up user files; and recovering and diagnosing system errors. This guide should be read in conjunction with **SETTING UP VENIX** which contains useful information concerning the installation of a VENIX system.

Throughout this document, references are made to VENIX commands which can be useful for system maintenance. The descriptions of these commands are intended to suggest their most common uses, but no attempt is made to explain all command features and options. The *User Reference Manual* contains a complete write-up of all the VENIX commands. Sections in the *Programmer Reference Manual* describe system calls, library functions, and file formats. Certain system programs which are never run directly by users (such as the clock daemon **cron**) are documented in **SYSTEM MAINTENANCE PROCEDURES**, section 8 in the *Installation and System Managers's Guide*.

2.1.1 The Super User and Super Group

The system administrator should use the **root** login so that he has super user capabilities. The super user can access any file without regard to permission settings, and can signal, terminate or suspend/resume any process, no matter who owns it. The super user can also execute certain privileged commands, such as **chown** (change the ownership of a file). Since the super user is free from most restrictions, he must take great care not to accidentally erase files or damage the system.

Users may obtain limited super capabilities by being in the 'super' group. Members of the super group have the ability to manipulate system files and directories (for example, to add or remove commands from the **/bin** directory). However, they have none of the special abilities the **root** has to access any user file regardless of permission setting, or to kill any process. (Super group members are also given permission to do certain things useful for real-time programming, such as run at very high priority.)

The user and group ID's are assigned for each user in the **/etc/passwd** file. **SETTING UP VENIX** describes how to edit this file to add new users. The super user has user ID of zero; the super group has group ID of zero. Normal users are given non-zero user ID's, and assigned to the default group called 'other' or placed in different groups as desired. Group names and group ID's are found in the file **/etc/group**.

2.2 MAINTAINING FILE SYSTEMS

2.2.1 Creating New File Systems

A 'file system' is a collection of files on a disk partition, internally structured so that VENIX can locate and manipulate individual files and directories. An empty file system must first be created on a disk partition before files can be used there. On a floppy diskette, a disk partition covers the entire physical unit. The hard disk, on the other hand, is divided into three or more partitions: the system area, temporary area, and one or more user areas.

When VENIX is initially transferred onto the hard disk, file systems are made for the system, user and temporary partitions. When you wish to use a floppy diskette as an extension of your hard disk file system, that is, to **mount** it on your directory tree, you must first make a file system on the diskette. You do not need to make a file system on diskettes which are to be used for back-up or transfer purposes with the **tar** or **dump** commands.

2.2.2 Making a File System

To make a file system on a disk partition, the /**etc/mkfs** command is used in the following format:

```
/etc/mkfs  name  length
```

where *name* is the name of the new partition and *length* is its length in blocks. (The sizes of all partitions are given in the device write-ups in section 7, **DEVICES**, in this manual.) For example, to create a new file system on RL02 unit /dev/rl0.usr, type:

```
/etc/mkfs /dev/rl0.usr 10240
```

Creating a new file system wipes out anything previously on the hard disk partition.

To make a file system on a diskette, place the diskette in the upper disk drive (f0 on the Micro-11, or rx0 for RX02 floppies on the PDP-11). Be sure that the diskette is seated properly, and that the arrow on the diskette is aligned with that on the drive. Close the door and type:

```
/etc/mkfs /dev/f0 790
```

for the Micro-11, or, for the PDP-11, type:

```
/etc/mkfs /dev/rx0 988
```

mkfs is the 'make a file system' command. The disk drive on the Micro-11 is specified as **/dev/f0** and the length of the file system on the floppy diskette is 790 blocks. The disk drive for RX02 floppies on the PDP-11 is specified as **/dev/rx0.** The length of the file system in blocks on the RX02 is 988 (single-sided) or 1976 (double-sided).

NOTE: Users of RX02 floppies on a PDP-11 system should substitute **/dev/rx0 988** for **/dev/f0 790** in the rest of the examples in this chapter.

mkfs erases all files (if any) on the diskette, and makes a file system. For this reason, you must be careful not to accidentally create a file system on a diskette containing valuable files.

If you wish to check for bad blocks on the diskette, use the **−b** flag, e.g.

```
/etc/mkfs −b /dev/f0 790
```

This will take a few minutes to run.

To make a file system on more diskettes, remove the diskette in drive 0, insert a new diskette, and re-type the **mkfs** command.

2.2.3 Mounting and Umounting

The **mount** command attaches a file system to a directory on the hard disk. As described in **SETTING UP VENIX**, file systems in the user partition of the hard disk are **mount**ed automatically by the command file **/etc/rc** when VENIX is brought into multi-user mode. Likewise, they are automatically **umount**ed (unmounted) when the system is brought back into single-user mode. It is entirely permissible to **mount** and **umount** file systems at any time that they are not in use. It is important to run **fsck** on the given file system to ensure its integrity and repair possible damage before mounting is done. Use of **fsck** is described later.

For example, the commands:

```
fsck /dev/rl1.usr
mount /dev/rl1.usr /u0
```

will first check partition **/dev/rl1.usr**, and then mount it on directory **/u0**.

The command:

```
umount /dev/rl1.usr
```

will unmount the system. It is crucial that the file system(s) on a disk are unmounted before the disk is physically unloaded. If this is not done, the file system will become corrupted.

If file systems are located on a floppy diskette, they must also be 'manually' **mounted** and **umounted**. Files systems on floppy diskettes are logically attached to the directory hierarchy by mounting them on an empty directory on the hard disk. Once this is done, the previously empty directory now accesses the root of the new file system; any files on that system now appear in that directory.

A file system must be created on the diskette before it is **mounted**. If a diskette already contains a file system, then go ahead with the **mount** command. If you are using a blank diskette, make a file system first, using the **mkfs** command. Before **mounting**, run an **fsck** to ensure that the file system is O.K. For example, the commands (substitute **/dev/rx0** for **/dev/f0** if you are using a PDP-11, instead of a Micro-11 system):

```
fsck /dev/f0
mount /dev/f0 /f0
```

will first check the file system on the floppy diskette (known as **/dev/f0** or **/dev/rx0**), and then **mount** that file system in a directory (**/f0** or **/rx0**) in the system area of the hard disk.

Once the file system is mounted on the hard disk, you can use any normal VENIX commands. Any of the file systems on the diskette are now a part of the regular system and user areas on the hard disk. You will find that the copy commands will enable you to copy files from the hard disk to the floppy diskette.

The command

```
umount /dev/f0
```

will unmount the system. Again, it is crucial that the file system(s) on a disk are **umounted** before the disk is physically unloaded!

If you remove a diskette without **umounting** it, you may have to reboot the system, which will run an **fsck** automatically to clean up the file system. When you attempt to re-mount the floppy diskette, first run an **fsck** before **mounting**.

Type:

```
fsck /dev/f0
```

Then use the **mount** command.

It is impossible to **mount** a file system if the given directory is in use (or is someone's current directory). It is also impossible to **umount** a file system if that system is in use. Type **cd /** to return to the root directory and then **umount**.

Mounting file systems is not considered a 'privileged' activity; any user may mount or unmount file systems so long as the given systems or directory are not in use.

Typing:

```
mount
```

alone will give you a list of currently mounted file systems. The **mount** listing is helpful if you are not sure if you've successfully **umount**ed the diskette. This list may occasionally disagree with reality or contain duplicate entries. This can happen after a crash or if file systems mounted while in single-user mode are not unmounted before going into multi-user mode. The list can be corrected by going into single-user mode, unmounting all file systems, and going again to multi-user mode.

If a disk pack is physically write-protected, or should not be written on, then it can be mounted 'read-only' by giving the **−r** flag after the directory name, as in:

```
mount /dev/rl1.usr /u0 -r
```

If a diskette is physically write-protected, or should not be written on, then it can be mounted 'read-only' with:

```
mount /dev/f0 /f0 -r
```

VENIX will prevent writing on any file or directory in file systems mounted this way. If a diskette is physically write-protected and mounted without the **−r** flag, errors messages will result even if you do not explicitly write on the diskette. This is because VENIX always tries to update the 'last-examined' date on files when they are read. The solution in this case is to simply **umount** the diskette and re-**mount** it 'read-only.'

2.2.4 Using fsck

fsck checks the file system on each partition of the hard disk, or a floppy diskette. An **fsck** of the system and users areas on the hard disks is done automatically. The disk partitions are listed in the file **/etc/checklist**.

You can run **fsck** on individual file systems by specifying the name of the partition as an argument, such as:

```
fsck /dev/rrl0.usr
```

To check the file system on the floppy diskette, run the **fsck** command, specifying the device name for the floppy diskette:

```
fsck /dev/f0
```

fsck checks all the blocks on the partitions, and lets the user know if any blocks are bad. **fsck** is talkative; it displays what is happening in some detail on the screen. After the **fsck** is run, you will see the output:

```
/dev/rw0.sys (system area)

Phase 1 - Check Blocks
Phase 2 - Not needed
Phase 3 - Check Pathnames
Phase 4 - Check Reference Counts
Phase 5 - Check Free List

/dev/rw0.usr (user area)

Phase 1 - Check Blocks
Phase 2 - Not needed
Phase 3 - Check Pathnames
Phase 4 - Check Reference Counts
Phase 5 - Check Free List
```

If **fsck** does find inconsistencies in a file system, it will usually ask you if you wish to fix things up. In general you can just type 'y' (yes); however, you should be aware that **fsck** may clean up the file system by removing all inconsistent files.

The 'adjusted link counts' and 'unreferenced files' found by **fsck** can be fixed with little danger of file loss, unless the directories leading to the file are damaged. The most frequent error is a 'bad free list' which can always safely be fixed. If any fixing is done, it is a good idea to run **fsck** a second time since in some cases not all errors will be fixed in just one run.

Sometimes damaged file systems will result in error messages calling some files 'BAD' (i.e., files with impossible block locations) and others 'DUP's (files containing blocks also found in other files). In Phase 3, always answer 'yes' if asked to remove a file previously marked BAD. If a file has just been reported to have DUP's previously, you might wish to avoid removing it immediately, since some of these duplicates may be cleared up when the BAD block files are deleted. A second **fsck** can then be run to see which files still contain duplicated blocks.

If you are running **fsck** and errors are reported, you have the option to copy important files before **fsck** cleans up the file system. When **fsck** asks for permission to clean up a specific file, you may type 'n' (no). Temporarily **mount** the file system 'read only' on the hard disk, e.g.:

```
mount /dev/f0 /f0 −r
```

and copy the file to another file system. Then **umount** the file system, and run **fsck** again on the diskette in the floppy drive.

If **fsck** reports errors and you give permission for fixing, you should immediately reboot the computer after **fsck** finishes.

Inconsistencies in file systems are rare, and are usually due to power failures or other sudden crashes which result in a halt of the computer without giving VENIX time to flush its internal I/O buffers. If you find that **fsck** frequently turns up problems with your hard disk or floppy diskette, then something may be seriously wrong with your disk hardware or device driver.

2.3 BACKING UP FILES

Files on a hard disk can be backed up by three methods:

- **mount**ing a diskette and copying the files
- using the **tar** (tape archive) command
- using the **dump** command

Both **tar** and **dump** write out information in their own particular formats, which are totally separate from the VENIX file system format. Diskettes written out by either program are not initialized with **/etc/mkfs** (make a file system). **mount**ed diskettes must have a file system.

If you **mount** and copy files to the floppy diskette, any previous iterations of a file are overwritten, and only your most recent version is saved. This saves space on the diskette. One disadvantage is that **mount**ing and copying is a lengthy process if you need to back up a number of files.

For most purposes, **tar** is the best method for backing up files. **dump** is provided only for compatibility with older versions of VENIX; its use is not recommended. **tar** provides all the functionality of **dump**, and considerably more flexibility, so we will not explain use of **dump** here. Interested users can examine the command write-up in the *User Reference Manual*.

2.3.1 Backing Up User Files

The most common way to lose user files is through accidental deletion by mistaken use of wildcard file names. Hardware failures can also wreak havoc with a file system. Since VENIX uses a tree-structured file system, it can be sensitive to bad blocks in key areas, so frequent backup is recommended.

The program **tar** can be used to backup user files. **tar** is the usual format for transferring files between UNIX, VENIX, and other UNIX-derived systems.

tar can be used with several options called up with flags. The flags are clustered together on the command line following 'tar'. Next, any arguments to these flags are listed in the order that the flags appear and separated by a space. The last item(s) appearing on the command line is the name(s) of the file or directory to be saved.

The **c** flag will clear the medium and store files on it. (The **c** is a mnemonic for 'create'.) Because the default device for **tar** is magnetic tape unit 1, the command:

```
tar c .
```

will initialize a new tape on unit 1 and save everything under the current directory (.). **tar** recursively descends subdirectories, so all directories and files beneath the given directory are found. So, to save everything on the system from root (/) down, on magtape unit 0, type:

```
tar c0 /
```

tar can be used on devices other than magtape. The **f** key is used to specify an alternative device. For instance, on the Micro-11, to store the directories 'cairo' and 'alexandria' on a diskette in the upper drive (**/dev/f0**), type:

```
tar cf /dev/f0 cairo alexandria
```

If you already have files on the diskette, use the **r** (restor) flag append new files to the diskette. To add the file 'sphinx' in the directory 'cairo' to a floppy diskette in the lower drive (**/dev/f1**), type:

```
tar rf /dev/f1 cairo/sphinx
```

If you use the '**r**' key, the diskette must already have been initialized with **tar c**.

Normally, **tar** does its work silently. The **v** (verbose) flag causes it to type the name of each file it treats preceded by the function letter. To verbosely add another directory **pyramids** to the end of the last file on the floppy on **/dev/f1**, type:

```
tar rvf /dev/f1 pyramids
```

Before running **tar**, the **du**(1) command can be used to determine the number of blocks beneath a given directory. (In calculating the number of blocks, be aware that **tar** uses one header block for each file saved.) If you expect the number of blocks being **tar**red to exceed the length of the medium, use the s flag on the command line followed by the number of blocks on the medium. A Micro-11 floppy contains 790 blocks. A single-sided RX02 contains 988 blocks while the double-sided contains 1976 blocks. Single density magtape contains 11 blocks per foot. Double density tape contains 22 blocks/foot. The figures for magnetic tape are larger when using raw tape and a large blocking factor.

To verbosely back up your system on 2400' single density magtape on unit 0, type:

```
tar cvs0 26400
```

tar with the **d** (date) flag allows backup of only those files modified after a specified number of days before the current date. To initialize an RX02 floppy in drive 1 holding only those files in **cairo** modified in the last seven days, type:

```
tar cdfs 7 /dev/rx1 988 cairo
```

Note how the **s** key is used here to tell **tar** the length of the RX02 diskette, so that it will prompt for and use additional diskettes if the first one becomes filled.

To simply view the files on the same floppy, type:

```
tar tvf /dev/rx1
```

To extract all files from the floppy:

```
tar xvf /dev/rx1
```

To verbosely extract a specific file in a directory, for example, file 'pyramids' in the directory 'cairo', from a tape in unit 1, type:

```
tar xv1 cairo/pyramids
```

tar always restores files using the same pathnames given when the files were saved. This means that full pathnames (that is, names beginning with a '/') should be avoided. For example, suppose that user Dennis has a home directory

```
/usr/dennis
```

and a directory beneath that:

```
/usr/dennis/letters
```

If directory **letters** is saved using the command

```
tar c /usr/dennis/letters
```

a subsequent extraction, with

```
tar x
```

will always bring back the old files to **/usr/dennis/letters,** overwriting any files of the same name in that directory. For this reason, it is desirable to save files according to a 'relative' name. For example, from Dennis' home directory (which is **/usr/dennis**):

```
cd          (change directory to /usr/dennis)
tar c letters
```

Now the command

```
tar x
```

will extract the saved files into whatever directory Dennis is in when he issues the command. So to restore the saved files to a temporary directory, the following commands could be used:

```
cd
mkdir temp
cd temp
tar x
```

Now the extracted files are placed in

/usr/dennis/temp/letters

while the directory

/usr/dennis/letters

is untouched. Of course, if Dennis really wishes to extract his files into the directory **/usr/dennis/letters,** and overwrite any copies of the same files on disk, he can always use the commands

```
cd
tar x
```

and this will happen.

2.3.2 Backing Up the System

SETTING UP VENIX describes the mechanism for duplicating your VENIX system on floppies or magnetic tape. If you haven't yet done this, do it now!

2.4 USEFUL COMMANDS AND FILES

2.4.1 Message of the Day

The message (if any) in the file **/etc/motd** (**m**essage **o**f **t**he **d**ay) is given whenever a user logs in. This is a convenient mechanism for sending users general notices (e.g. 'disk space low — clean up your files!').

2.4.2 The /etc/wall Command

The system administrator can communicate to all users through **/etc/wall** (**w**rite to **all**) which acts like **write**(1), but sends the message to every user logged in.

2.4.3 The ps Command

The **ps**(1) command will give a list of all processes running on the system, as well as their process ID's, current state (running, waiting, swapped, etc.) and CPU consumption.

2.4.4 The kill Command

The **kill** command can be used to stop a process. Each process has an ID number (pid), which you will see listed when you run the **ps** command. For example, the command:

```
kill 1197
```

will send a 'terminate' signal to the process with ID number '1197'. This is enough to finish most processes. To be sure, the command:

kill −9 *process ID*

will always terminate the process.

If you don't remember the process ID number, or don't want to take time to get a display status (perhaps your printer is going haywire), you can type:

```
kill 0
```

This will terminate all processes, including those which are running in the background.

2.4.5 The suspend and resume Commands

The **suspend**(1) command will suspend a process with given ID; the process will remain frozen until it is **resumed**. Any user can **suspend** or **resume** his or her own processes; the super-user can do it to any process.

2.4.6 The chmod Command

chmod can be used to change the access permissions on files. The command:

chmod 660 *file1 file2*

changes the permission setting on *file1* and *file2* to the octal value '660'. The three digits '660' represent the values for the user, group, and other permission, with each digit encoding the read/write/execute permission.

4	**2**	**1**	**4**	**2**	**1**	**4**	**2**	**1**
read	*write*	*execute*	*read*	*write*	*execute*	*read*	*write*	*execute*
	U S E R			**G R O U P**			**O T H E R**	

The value '660', then, corresponds to read/write permission for user and group, and no permission for others. When a long list of the file is done (**l** command), the permission setting will be seen as:

```
rw-rw---
```

It is also possible to indicate the modes symbolically. For example, the command

```
chmod o+r file1
```

adds to category other (**o**) the permission to read (**r**).

2.4.7 The chown and chgrp Commands

The super user can change user and group file ownerships. **chown** is used to change the ownership of a file. For example, the command:

```
chown max file.a
```

sets the ownership of **file.a** to user **max.**

The **chgrp** command changes the group ownership of a file. The command:

```
chgrp biolab file.a
```

changes the group ownership of **file.a** to group **biolab.**

The **l** command will list information on any file, including user and group ownership. For example, if you type:

```
l file.a
```

VENIX will print the access permission settings, and then the user ownership (plus other information about the file):

```
-rw-rw-r-- 1 max    4639 Aug 1 13:24 file.a
```

l −g file.a will do the same, but the group ownership of the file will be listed:

```
-rw-rw-r-- 1 biolab 4639 Aug 1 13:24 file.a
```

User and group ownerships are specified in the **/etc/passwd** file along with the user name and directory. Group ID's and names are listed in the **/etc/group** file.

2.4.8 The find Command

find is a versatile command that can be used to locate files, beneath a directory, that match any of a variety of conditions. For example:

```
find /usr −size +100 −atime +30 −print
```

will list all files under directory **/usr** which are greater than 100 blocks in size and have not been accessed in thirty days.

2.4.9 Clock Daemon

When VENIX comes up, the clock daemon **cron** may be started up by a command in **/etc/rc**. As distributed, however, the **cron** command is commented out. The comment colon needs to be removed to enable **cron** to work.

cron sleeps in the background, and wakes up periodically to examine the file **/usr/lib/crontab** which contains a list of commands and when to execute them. (The frequency at which **cron** checks the file is given by the first command argument, and is usually once every 10 minutes)

crontab normally contains a reference to run the command **/usr/lib/atrun** every half hour, which in turn executes commands that users have scheduled through **at**(1). The frequency of **atrun**'s execution can be adjusted here. Also, additional lines can be added to **crontab** to regularly run more commands at certain hours, days of the week or month. (Commands which need only be run once can be scheduled with **at**; commands which must be run regularly should be run directly from **crontab.**) See **cron** in **SYSTEM MAINTENANCE PROCEDURES**, section 8 in this manual.

cron is really very helpful. For an example of how **cron** functions 'behind the scenes', the following line could be placed in **/usr/lib/crontab** to invoke **find** at 4am every morning to remove all files whose names being with 'junk' that have not been accessed in thirty days:

```
0 4 * * * find /usr -name "junk*" -atime +30 -exec rm { } \;
```

2.5 SYSTEM ACCOUNTING

2.5.1 Disk Space

Within each file system, VENIX itself cannot restrict the number of disk blocks available to any particular user: any user may extends his or her directories and create new and larger files until the total amount of space available in that file system is reached. When there is no room left on a file system, programs will complain of I/O errors and an 'Out of space' message will appear on the console terminal. This will continue until someone removes some files, or until the system administrator removes some.

Listed below are several utilities which are useful for managing disk space. Some of these commands, such as **df**, are applied to entire disk partitions by their '/**dev**' name. Others, such as **du**, are applied to named directories. All of these are described in section one of the *User Reference Manual* along with additional options.

2.5.2 The df Command

df(1) ('**disk free**') determines the amount of free blocks on the system or user partition. For example:

```
df /dev/rl0.usr
```

or, for winchester users:

```
df /dev/w0.usr
```

will give a free block count on the user area.

2.5.3 The quot Command

The **quot**(1) command lists the number of blocks per user on the system or user area.

```
quot /dev/rl0.usr
```

will give such a list of blocks for each user. (Substitute **/dev/w0.usr** on command line if using a Micro-11.)

2.5.4 The du Command

du(1) ('**d**isk **u**sage') determines the number of blocks used under a particular directory, and can give totals for each sub-directory.

```
du /usr/letters/auntie
```

lists the number of blocks used by all files and totals for all subdirectories under **/usr/letters/auntie**.

2.5.5 The ncheck Command

ncheck(1) lists all files and directories on the hard disk or a floppy diskette. This is helpful when you wish to see all the directories listed. (The **l** command lists only the contents of a directory.) For example:

```
ncheck /dev/rl0.sys
```

checks the system area; on a winchester,

```
ncheck /dev/rl0.sys
```

checks the system area.

To check an RX02 floppy diskette:

```
ncheck /dev/rx0
```

To check a Micro-11 floppy:

```
ncheck /dev/f0
```

2.6 LOGIN TIME

2.6.1 The ac Command

The **ac**(1) utility can be used to list the amount of terminal time taken by each user, as collected in the **/usr/adm/wtmp** file. The command

```
ac −p
```

will print totals for each user; see **ac**(1) for other options.

Accounting will only be done if the **/usr/adm/wtmp** file exists, so the system administrator should create an empty **wtmp** if accounting of this type is desired. This can be done with the command

```
cp /dev/null /usr/adm/wtmp
```

Once this file is created, it will grow *ad infinitum*, so relevant information should periodically be collected with **ac** and the file cleared.

2.6.2 The who Command

The **who**(1) command with no arguments will give a list of all users currently logged in, and the time they logged in. The command

```
who /usr/adm/wtmp
```

will give the precise times each user logged in and out over the history of the **wtmp** file.

2.7 SYSTEM FILES AND DIRECTORIES

The following is a quick overview of the system directories and files:

/	root	
/dev/		device nodes
	console	main console
	rl*	
	rm*	
	rp*	
	rx*	
	rrl*	raw disks
	tty??	terminals
	w0.*	winchester
	...	
	arl*	asynchronous raw disks
	...	
/bin/		utility programs, cf /usr/bin (1)
	as	assembler
	cc	C compiler executive, cf /usr/lib/c0
	...	
/lib/		object libraries and other stuff, cf /usr/lib
	libc.a	system calls, standard I/O, etc. (2,3,3S)
	libm.a	math routines (3M)
	libplot.a	plotting routines, **plot**(3)
	c[012]	pass of **cc**(1)
	...	
/etc/		essential data and dangerous maintenance utilities
	passwd	password file (4)
	group	group file (4)
	motd	message of the day, **login**(1)
	mtab	mounted file table, **mtab**(4)
	dtab	dump history, **dump**(1)
	ttys	properties of terminals, **ttys**(4)
	getty	part of login, **getty**(8)
	init	the father of all processes, **init**(8)
	rc	shell program to bring system up
	cron	the clock daemon, **cron**(8)

	...		
/tmp/			temporary files, cf /usr/tmp
	e*		used by **ed**(1)
	pc*		used by **cc**(1)
	...		
/usr/			mounted user file system, general purpose directory
	demo/		demonstration
	guest/		guest login directory
	adm/		administrative information
	bin/		utility programs
	dict/		hash tables, etc. for **spell**
	games/		games
	...		
	include/		standard #include files
		a.out.h	object file layout, a.out(4)
		stdio.h	standard I/O, **stdio**(3)
		math.h	(3M)
		...	
		sys/	system-defined layouts
		...	
	lib/		object libraries to keep /lib/ small
		atrun	scheduler for **at**(1)
		struct/	passes of **struct**(1)
		tmac/	macros for **nroff**(1)
		suftab	table of suffixes for **nroff**(1)
		...	
/u?/			other mounted user file systems
	...		
/venix			bootable image of the kernel
/etc/.profile			login profile for root
/.profile			boot profile

The **dtree**(1) command can be used to generate a complete list of system names.

2.7.1 File Modes

Most system commands and other files are given read/execute permission to everyone, and write permission to the root only. Certain commands which need to be accessible by users, but also do privileged things, are given the "set UID" mode. When these commands are executed by users, the effective ID is made "root" so that privileged things can be done. At the time of this writing the following files were set-UID:

/bin/ps	**/bin/mkdir**	**/bin/su**
/bin/mount	**/bin/umount**	**/bin/rmdir**
/bin/login	**/bin/mv**	**/bin/at**
/bin/rmail	**/bin/newgrp**	**/bin/lpstop**
/bin/mail	**/bin/passwd**	**/usr/lib/atrun**
/usr/lib/uucp/uucico	**/usr/lib/uucp/uuxqt**	

These programs will not run correctly without set-UID mode. If these files' permissions are listed (with **ls** or **l**), an s will be shown in place of the "user" **x** bit. The command

```
chmod u+s file
```

will make **file** a set-UID program;

```
chmod u-s file
```

will turn off set-UID action.

Commonly used programs have another special mode bit, called the "sticky" bit. "Sticky" programs are saved permanently ("stick around") in the swap area upon their first use (after the system is booted). They can subsequently be run slightly faster than normal, since they can be quickly found in the swap area. At time of writing, the following commands were among those made "sticky":

/bin/sh	**/bin/l**	**/bin/vi**	**/lib/c0**
/bin/ps	**/bin/ls**	**/bin/ice**	**/lib/c1**

If these files' permissions are listed, a **t** will be shown in place of the "others" **x** bit.

The sticky bit is only in effect if the programs are pure (type 410), so that the code segment can be shared by multiple users. This corresponds to the **−n** flag of **cc**(1) or **ld**(1). (Even if the sticky bit is not set, programs of this type will have their code segment shared automatically if they are executed simultaneously by two or more users.)

One special rule about "sticky" commands: after they are run once, they should never be removed or replaced, or the file system will become slightly corrupted. If you need to remove/replace such a file, you should rename the file to something else (e.g. "file.old") and remove it after the system is next rebooted.

2.7.2 File Links

Certain system files are linked more than once; that is, there is more than one name corresponding to those files. For example, the **l** and **ls** files are linked together; the actual program checks the second letter of its name (present in **argv[0]**) to determine whether it was called "ls" or "l", and acts appropriately. The user can create his or her own links with the **ln**(1) command. The link must be within a mounted file system.

The following files were linked together at the time of this writing:

/bin/l	**linked with**	**/bin/ls**
/bin/mount	″	**/etc/mount**
/bin/umount	″	**/etc/umount**
/bin/mail	″	**/bin/rmail**

2.8 SYSTEM ERRORS AND CRASHES

There are two types of console error messages. Recoverable console messages indicate some difficulty which the administrator should be aware of, but do not result in a system-wide stoppage. Messages beginning with ″PANIC: ...″ indicate unrecoverable system errors, and result in the system coming to an abrupt halt. (Time is taken to flush the system buffers, however, so the integrity of the file system is usually maintained.)

When restarting after a crash, bring the system up single-user and perform an **fcheck**(1) on all file systems which could have been in use at the time of the crash (see the section on **fcheck** earlier in this document). If there are any problems, they should be repaired. When you are satisfied with the health of your disks, set the date and come up multi-user as usual by typing CTRL-D.

To even boot VENIX at all, three files (and the directories leading to them) must be intact. First, the initialization program **/etc/init** (see **init** in appendix A) must be present and executable. If it is not, the CPU will loop in user mode at location 6. For **init** to work correctly, **/dev/console** and **/bin/sh** must be present. If either does not exist, the system begins thrashing: **init** will go into a fork/exec loop trying to create a Shell with proper standard input and output.

If you cannot get the system to boot, you will have to use a backup copy to run on. The damaged system may then be repaired with **fcheck** as described earlier. If serious surgery is required on the root file system, it is prudent to simply discard it and return to the most recent backup system available, since files may have been insidiously corrupted. If necessary, you can try to salvage any important files modified since the last backup was made.

2.8.1 Recoverable Errors

There are a number of situations which will cause these kind of messages to appear on the console terminal. In the following messages, expressions of the form "on dev A/B" indicate that the error occurred on the device with major and minor numbers **A** and **B**, respectively. You can find out the name of this device by using the major number as an index into the block or character table in the **/usr/sys/conf/c.i** file (it is usually clear whether the error was due to the block or character device with the given number). The minor number can be discovered from looking at the driver write-up in section 4 of the *VENIX Reference Manual.* A long listing of the **/dev** directory, as in

```
l /dev
```

will also provide the major and minor numbers corresponding to each entry.

The following is a list of recoverable system error messages.

Error on dev A/B

This is the most common kind of error message and is produced by a device driver. Typical causes of this message are an attempt to read a disk or tape which is not loaded or is off-line, or an attempt to write on a protected disk or tape. This message is followed by a line of the form

bn X er Y Z

X is the block number where the error occurred; Y and Z are device register values, typically the error and control/status registers. See the DIAGNOSTICS section of the device driver write-up in section 7 (or check the device driver source for occurrences of the **deverror( )** call) to see which device register values are being printed. The "bn" message prints out the block number and two register values in the order passed to this function. Once you have determined which device register is being printed, you can check your hardware manual to see what the various bits mean.

No space on dev A/B

All blocks are used up on the device with major/minor numbers **A** and **B**. See section on disk space.

Bad block on dev A/B

A block was found with number placing it outside the bounds of the file system. Run **fcheck** to delete the file with the bad block and fix up the file system.

No inodes on dev A/B

All the inode pointers for the file system on the device with major/minor numbers **A** and **B** have been used up. This should only occur if you have many small files on a file system. When a file system is made, the **/etc/mkfs** command automatically allocates enough inodes to handle most disk requirements. If this is insufficient, you can clear away all files and make a new file system. **/etc/mkfs** allows the user to specify a "prototype" file system giving the number of inodes to be allocated.

Bad count on dev A/B

An invalid freelist count or invalid inode count was found on the super block of the device with major/minor numbers **A** and **B**. At this point, all the counts are zeroed which will almost certainly lead to a "No space" diagnostic. Use **fcheck**(1) to repair the bad file system.

No swap

A program or shared segment needs to be swapped out, and there is no swap space left on disk. If this error occurs regularly, the swap area should be increased: see "Setting Up VENIX".

No shared

A shared text (pure procedure) program is being executed or shared data segment created, and the table for such things is full. The maximum number of shared segments is set by the NSEG parameter in **param.h** file; see *"Setting Up VENIX"*.

Inodes full

The in-core inode table is full, causing an open( ) call to fail. The maximum number of in-core inodes is set by the NINODE parameter in the **param.h** file; see "Setting Up VENIX" for adjusting this parameter.

Tout ovflw

The in-core timeout table (used by fast alarms, etc.) is full and a timeout call has been lost. The maximum number of timeouts is set by the NCALL parameter in the **param.h**. See "Setting of VENIX" for adjusting this parameter.

No files

The in-core file table (one for each open file on the system) is full, causing an open( ) call to fail. The maximum number of files is set by the NFILE parameter in the **param.h** file; see "Setting of VENIX" for adjusting this parameter.

Interrupt at X

An unexpected interrupt has occurred through interrupt vector *X* (octal) modulo 0100. Only the low 6 address bits are significant. Either the **l.s** table is set up incorrectly or the hardware has generated a bogus interrupt.

2.8.2 Unrecoverable Errors (Panics)

The following messages will appear after the word "PANIC", and are followed by a total system hang-up. Most of these messages will not be useful to most users since many of these problems can only be fixed by modifying the system libraries or tables, which requires a VENIX system configuration kit.

Bdev The **getblk** routine was called with a nonexistent major device as an argument. Check the "rootdev," "swapdev," and "pipedev" assignments in the **config** file (see *"Setting Up VENIX"*).

Dtab Null device table entry for the major device used as an argument to **getblk**.

Iinit An I/O error reading the super-block for the root file system during initialization. Check the "rootdev," "swapdev," and "pipedev" assignments in the **config** file (see *"Setting Up VENIX"*).

No fs A device has disappeared from the mounted-device table. (Due only to hardware or software bug.)

Swap error
An unrecoverable I/O error during a swap. Really shouldn't be a panic, but it's hard to fix.

Unlink The directory containing a file being deleted can't be found.

Trap An unexpected trap has occurred within the system. This is accompanied by two numbers: a 'pc' which was the value (in octal) of the program counter when the trap took place; and a 'trap type' which encodes which trap occurred. The trap types are:

0	**bus error**
1	**illegal instruction**
2	**BPT/trace**
3	**IOT**
4	**power fail**
5	**EMT**
6	**recursive system call (TRAP instruction)**
7	**programmed interrupt**
8	**floating point trap**
9	**segmentation violation**
10	**cache parity error**

In some of these cases it is possible for 16 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred. On non-split I/D kernels, a symbol table listing of the kernel can be obtained with the command

```
nm -ng /venix
```

Users with VENIX configuration kits can use this table to locate the routine in which the trap occurred.

Chapter 3
UUCP IMPLEMENTATION DESCRIPTION

3.1 INTRODUCTION

Uucp is a series of programs designed to permit communication among UNIX and VENIX systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. This paper describes the second implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by uucp daemons. There are three types of files used for the execution of work. Data files contain data for transfer to remote systems. Work files contain directions for file transfers among systems. Execution files are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp	This program creates work and gathers data files in the spool directory for the transmission of files.
uux	This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.
uucico	This program executes the work files for data transmission.
uuxqt	This program executes the execution files for UNIX command execution.

The secondary programs are:

uulog	This program updates the log file with new entries and reports on the status of uucp requests.
uuclean	This program removes old files from the spool directory.

The remainder of this chapter describes the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

3.2 UUCP — UNIX TO UNIX FILE COPY

The **uucp** command is the user's primary interface with the system. The **uucp** command was designed to look like **cp** to the user. The syntax is:

uucp [*option*] ... *source* ... *destination*

where the *source* and *destination* may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by **uucp** are:

−d	Make directories when necessary for copying the file.
−c	Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.
−g*letter*	Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)
−m	Send mail on completion of the work.
	The following options are used primarily for debugging:
−r	Queue the job but do not start the uucico program.
−s*dir*	Use directory *dir* for the spool directory.
−x*num*	*Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source's name may contain special shell characters such as "?*[]". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

where the command-string is made up of one or more arguments. All special shell characters such as "<>|^" must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain an "!" will not be treated as files. (They will not be copied to the execution machine.) The "−" is used to indicate that the standard input for *command-string* should be inherited from the standard input of the **uux** command. The options, essentially for debugging, are:

−r	Don't start uucico or uuxqt after queuing the job;
−x*num*	*Num* is the level of debugging output desired.

The command:

```
pr abc | uux - usg!lpr
```

will set up the output of "pr abc" as standard input to an "lpr" command to be executed on system "usg".

uux generates an execute file which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, **uux** will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by the uucico program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The execute file will be processed by the uuxqt program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below:

3.3.1 User Line

U *user system*

where the *user* and *system* are the requester's login name and system.

3.3.2 Required File Line

F *file-name real-name*

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the execute file. The uuxqt program will check for the existence of all required files before the command is executed.

3.3.3 Standard Input Line

I *file-name*

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the **uux** command if the "–" option is used. If a standard input is not specified, **/dev/null** is used.

3.3.4 Standard Output Line

O *file-name system-name*

The standard output is specified by a ">" within the command-string. If a standard output is not specified, **/dev/null** is used. (Note: the use of ">>" is not implemented.)

3.3.5 Command Line

C *command* [*arguments*] ...

The *arguments* are those specified in the command-string. The standard input and standard output will not appear on this line. All required files will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell. In addition, a shell "PATH" statement is prepended to the command line as specified in the uuxqt program.

After execution, the standard output is copied or set up to be sent to the proper place.

3.4 UUCICO — COPY IN, COPY OUT

The uucico program will perform the following major functions:

- — Scan the spool directory for work.
- — Place a call to a remote system.
- — Negotiate a line protocol to be used.
- — Execute all requests from both systems.
- — Log work requests and work completions.

Uucico may be started in several ways:

a) by a system daemon,

b) by one of the uucp, uux, uuxqt or uucico programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the ''shell'' field in the **/etc/passwd** file for the uucp logins.)

When started by method a, b or c, the program is considered to be in MASTER mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in SLAVE mode.

The MASTER mode will operate in one of two ways. If no system name is specified (−s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The uucico program is generally started by another program. There are several options used for execution:

−r1	Start the program in MASTER mode. This is used when uucico is started by a program or ''cron'' shell.
−s*sys*	Do work only for system *sys*. If −s is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

–d*dir*	Use directory *dir* for the spool directory.
–x*num*	*Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the uucico program.

3.4.1 Scan For Work

The names of the work related files in the spool directory have format:

type . system-name grade number

where:

Type	is an upper case letter, (C= copy command file, D= data file, X= execute file);
System-name	is the remote system;
Grade	is a character;
Number	is a four digit, padded sequence number.

The file

C.res45n0031

would be a work file for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for work files (files with prefix "C."). A list is made of all systems to be called. Uucico will then call each system and process all work files.

3.4.2 Call Remote System

The call is made using information from several files which reside in the **uucp** program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the **L.sys** file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number if field [3] is ACU or the device name (same as field [3]) if not ACU,

[6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The phone number may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the **L-dialcodes** file.

The **L-devices** file is scanned using fields [3] and [4] from the **L.sys** file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of uucico will not try to use it. If the call is complete, the login information (field [6] of *L.sys*) is used to login.

The conversation between the two uucico programs begins with a handshake started by the called, SLAVE system. The SLAVE sends a message to let the MASTER know it is ready to receive the system identification and conversation sequence number. The response from the MASTER is verified by the *SLAVE* and if acceptable, protocol selection begins. The SLAVE can also reply with a ''call-back required'' message in which case, the current conversation is terminated.

3.4.3 Line Protocol Selection

The remote system sends a message

P*proto-list*

representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a use-protocol message. The use-protocol message is

U*code*

where *code* is either a one character protocol letter or **N** which means there is no common protocol.

3.4.4 Work Processing

The initial roles (MASTER or SLAVE) for the work processing are the mode in which each program starts. (The MASTER has been specified by the **−r1** uucico option.) The MASTER program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are:

S	send a file,
R	receive a file,
C	copy complete,
X	execute a uucp command,
H	hangup.

The MASTER will send **R**, **S** or **X** messages until all work from the spool directory is complete, at which point an **H** message will be sent. The SLAVE will reply with **SY**, **SN**, **RY**, **RN**, **HY**, **HN**, **XY**, **XN**, corresponding to yes or no for each request.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message **CY** will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a **CN** message is sent. (In the case of **CN**, the transferred file will be in the spool directory with a name beginning with "TM".) The requests and results are logged on both systems.

The hangup response is determined by the SLAVE program by a work scan of the spool directory. If work for the remote system exists in the SLAVE's spool directory, an **HN** message is sent and the programs switch roles. If no work exists, an **HY** response is sent.

3.4.5 Conversation Termination

When a **HY** message is received by the MASTER it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final "OO" message to the other. The original SLAVE program will clean up and terminate. The MASTER will proceed to call other systems and process work as long as possible or terminate if a −s option was specified.

3.5 UUXQT — UUCP COMMAND EXECUTION

The uuxqt program is used to execute execute files generated by uux. The uuxqt program may be started by either the uucico or uux programs. The program scans the spool directory for execute files (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The execute file is described in the "Uux" section above.

3.5.1 Command Execution

The execution is accomplished by executing a **sh −c** of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

3.6 UULOG — UUCP LOG INQUIRY

The uucp programs create individual log files for each program invocation. Periodically, uulog may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The uulog program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

−s*sys*	Print entries where *sys* is the remote system name;
−u*user*	Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

3.7 UUCLEAN — UUCP SPOOL DIRECTORY CLEANUP

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

–d*dir*	The directory to be scanned is *dir*.
–m	Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)
–n*hours*	Change the aging time from 72 hours to *hours* hours.
–p*pre*	Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
–x*num*	This is the level of debugging output desired.

3.8 SECURITY

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the uucp system.

— The login for uucp does not get a standard shell. Instead, the uucico program is started. Therefore, the only work that can be done is through uucico.

— A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be setup to require call-back for certain login-ids. (See the"Files required for execution" section for the file description.)

— A conversation sequence count can be set up so that the called system can be more confident that the caller is who he or she says he/she is.

— The uuxqt program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specified in the uuxqt program. The installer may modify the list or remove the restrictions as desired.

— The **L.sys** file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

3.9 UUCP INSTALLATION

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local uucp *system-name*. Changes to these sources, however, can be done only by installations with UNIX source licenses. Except where noted, installations with binary-only licenses must use the predefined default names.

The four directories are:

lib	**(/usr/src/cmd/uucp)** This directory contains the source files for generating the uucp system.
program	**(/usr/lib/uucp)** This is the directory used for the executable system programs and the system files.
spool	**(/usr/spool/uucp)** This is the spool directory used used during uucp execution.
xqtdir	**(/usr/spool/uucp/.XQTDIR)** This directory is used during execution of execute files.

The names given in parentheses above are the default values for the directories. The italicized names *lib*, *program*, *xqtdir* and *spool* will be used in the following text to represent the appropriate directory names.

The file **/usr/include/ident.h** may be modified to change the installation name uucp uses. This file is read by uucp at run-time (even though it appears to be a C source file); therefore, even those installations possessing a binary-only UNIX license may change their uucp name.

There are two files which may require modification if the installation holds a source license: the **makefile** file and the **uucp.h** file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

3.9.1 Uucp.h Modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

3.9.2 Makefile Modification

There are several **make** variable definitions which may need modification.

INSDIR	This is the *program* directory (e.g. INSDIR = /usr/lib/uucp). This parameter is used if "make cp" is used after the programs are compiled.
IOCTL	This is required to be set if an appropriate **ioctl** interface subroutine does not exist in the standard "C" library; the statement "IOCTL = ioctl.o" is required in this case.
PKON	The statement "PKON = pkon.o" is required if the packet driver is not in the kernel.

3.9.3 Compile the System

The command:

```
make
```

will compile the entire system. The command

```
makecp
```

will copy the commands to the appropriate directories.

The programs uucp, uux, and uulog should be put in "**/usr/bin**". The programs uuxqt, uucico, and uuclean should be put in the *program* directory.

3.9.4 Files Required for Execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

3.9.5 L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by uucp. The special device files are assumed to be in the **/dev** directory. The format for each entry is:

line call-unit speed

where:

line	is the device for the line (e.g. cul0),
call-unit	is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number "0" in this field.),
speed	is the line speed.

The line:

```
cul0  cua0  300
```

would be set up for a system which had device "cul0" wired to a call-unit "cua0" for use at "300" baud.

3.9.6 L-dialcodes

This file contains entries with location abbreviations used in the **L.sys** file (e.g. py, mh, boston). The entry format is:

abb dial-seq

where;

abb	is the abbreviation,
dial-seq	is the dial sequence to call that location.

The line:

```
py  165–
```

would be set up so that entry "py7777" would send "165–7777" to the dial-unit.

3.10 LOGIN/SYSTEM NAMES

It is assumed that the login name used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique system name which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

3.10.1 USERFILE

This file contains user accessibility information. It specifies four types of constraint:

[1] which files can be accessed by a normal user of the local machine,

[2] which files can be accessed from a remote computer,

[3] which login name is used by a particular remote computer,

[4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format:

login,sys [*c*] path-name [*path-name*] ...

where:

login	is the login name for a user or the remote computer,
sys	is the system name for a remote computer,
c	is the optional call-back required flag,
path-name	is a path-name prefix that is acceptable for user.

The constraints are implemented as follows:

[1] When the program is obeying a command stored on the local machine, MASTER mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a null login name is used.

[2] When the program is responding to a command from a remote machine, SLAVE mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a null system name is used.

[3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a null system name.

[4] If the line matched in [3] contains a "c", the remote machine is called back before any transactions take place.

The line:

```
u,m  /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

```
dan,  /usr/dan
```

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan".

The lines:

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool".

The lines:

```
root,  /
,  /usr
```

allows any user to transfer files beginning with "/usr" but the user with login root can transfer any file.

3.10.2 L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

3.10.2.1 system name

The name of the remote system.

3.10.2.2 time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800 – 1730).

The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be **Wk** for any week-day or **Any** for any day.

The time should be a range of times (e.g. 0800 – 1230). If no time portion is specified, any time of day is assumed to be ok for the call.

3.10.2.3 device

This is either ACU or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

3.10.2.4 speed

This is the line speed for the call (e.g. 300).

3.10.2.5 phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the **L-dialcodes** file (e.g. mh5900, boston995 – 9980).

For the hardwired devices, this field contains the same string as used for the device field.

3.10.2.6 login

The login information is given as a series of fields and subfields in the format

expect send [*expect send*] ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The *expect* field may be made up of subfields of the form:

expect[– send – expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string "EOT" will send an EOT character and the string "BREAK" will try to send a BREAK character. (The BREAK character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the **L.sys** file would be:

```
sys Any ACU 300  mh7654 login uucp ssword: word
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

3.11 ADMINISTRATION

This section indicates some events and files which must be administered for the uucp system. Some administration can be accomplished by shell files which can be initiated by crontab entries. Others will require manual intervention. Some sample shell files are given toward the end of this section.

3.11.1 SQFILE — Sequence Check File

This file is set up in the program directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most recent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

3.11.2 TM — Temporary Data Files

These files are created in the spool directory while files are being copied from a remote machine. Their names have the form

TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of uucico and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the uuclean program is useful in this regard. The command:

```
uuclean  -pTM
```

will remove all *TM* files older than three days.

3.11.3 LOG — Log Entry Files

During execution of programs, individual *LOG* files are created in the spool directory with information about queued requests, calls to remote systems, execution of **uux** commands and file copy results. These files should be combined into the *LOGFILE* by using the uulog program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command:

```
uulog
```

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new *LOG* entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the uulog program will not read or remove them. To remove them, either use **rm**, **uuclean**, or change the mode to 0666 and let **uulog** merge them with the *LOGFILE*.

3.11.4 STST — System Status Files

These files are created in the spool directory by the uucico program. They contain information of failures such as login, dialup or sequence check and will contain a TALKING status when two machines are conversing. The form of the file name is

STST.*sys*

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a TALKING status. In this case, the file must be removed before a conversation is attempted.

3.11.5 LCK — Lock Files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is:

LCK..*str*

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

3.12 SHELL FILES

The uucp program will spool work and attempt to start the uucico program, but the starting of uucico will sometimes fail. (No devices available, login failures etc.). Therefore, the **uucico** program should be periodically started. The command to start uucico can be put in a "shell" file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands:

```
    program /uulog
    program /uucico  -r1
```

Note that the **−r1** option is required to start the uucico program in MASTER mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C*. or *D*. files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
    program /uuclean  -pTM -pC. -pD.
    program /uuclean  -pST -pLCK -n12
```

can be used. Note the **−n12** option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the **−n** option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like:

```
    cp spool /LOGFILE  spool /o.LOGFILE
    rm spool /LOGFILE
```

can be used.

3.13 LOGIN ENTRY

One or more logins should be set up for uucp. Each of the **/etc/passwd** entries should have the *program*/uucico as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving files, it should be given as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the shell argument limits the login to the use of uucp (uucico) only.

3.13.1 File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs uucp, uux, uucico and uuxqt should be owned by the uucp login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard shell for the uucp logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the program directory should be owned by the uucp login and set with mode 0400.

Chapter 4
A DIAL-UP NETWORK OF UNIX SYSTEMS

4.1 SYSTEM OPERATION AND DESIGN

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, **uucico**, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. **uucico** then selects a device and establishes the connection, logs onto the remote machine and starts the **uucico** program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites (systems that do not have the hardware to initiate a connection), however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this network among independent sites, all of which store proprietary programs and data, illustrates the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use a password to log into the local machine and then the local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses is identified. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and can require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

4.2 PROCESSING

The user has two commands which set up communications, **uucp** to set up file copying, and **uux** to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by **uucp** daemons.

4.2.1 File Copy

The **uucico** program is used to perform all communications between the two systems. It performs the following functions:

[1] Scan the spool directory for work.

[2] Place a call to a remote system.

[3] Negotiate a line protocol to be used.

[4] Start program **uucico** on the remote system.

[5] Execute all requests from both systems.

[6] Log work requests and work completions.

uucico may be started in several ways;

a) by a system daemon,

b) by one of the **uucp** or **uux** programs,

c) by a remote system.

4.2.2 Scan For Work

The file names in the spool directory are constructed to allow the daemon programs (**uucico, uuxqt**) to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

4.2.3 Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same *phone*number to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of **uucico** will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the **uucico** program. The conversation between the two **uucico** programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

4.2.4 Line Protocol Selection

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where *code* is either a one character protocol letter or an *N* which means there is no common protocol.

4.2.5 Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S	**send a file,**
R	**receive a file,**
C	**copy complete,**
H	**hangup.**

The *MASTER* will send **R** or **S** messages until all work from the spool directory is complete, at which point an **H** message will be sent. The *SLAVE* will reply with **SY**, **SN**, **RY**, **RN**, **HY**, **HN**, corresponding to **yes** or **no** for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message **CY** will be sent if the VENIX **cp** command, used to copy from the spool directory, is successful. Otherwise, a **CN** message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a **HN** message is sent and the programs switch roles. If no work exists, an **HY** response is sent.

4.2.6 Conversation Termination

When an **HY** message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4.3 PRESENT USES

One application of this software is remote mail. Normally, a VENIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

A primary use of a network is software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. It's a good idea to implement a "stockroom" which allows remote users to call in and request software. Also a "stock list" of available programs, and new bug fixes and utilities should be added regularly. In this way, users can always obtain the latest version of any program. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, **uucp** does remote-to-remote copies.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility has been the **diff** program which compares two text files and indicates the differences, line by line, between them. Only lines which are not identical are printed. Similarly, the program **uudiff** compares files (or directories) on two machines. One of these directories may be on a passive system.

To avoid moving large numbers of usually identical files, **uudiff** computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The **uux** command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The **uux** command allows the formatting of the printout on the local machine and printing on the remote machine using standard VENIX command programs.

4.4 PERFORMANCE

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

4.5 SYSTEM GOALS

A full system of remote software maintenance is a network goal. The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, systematic regression testing should be done both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services should be implemented, like inter-system **mail** and **diff**, plus the many implied commands represented by **uux**. Also there's a need for inter-system **write** (real-time inter-user communication) and **who** (list of people logged in on different systems). A slow-speed network of this sort is very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, is dependent on the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

Chapter 5

THE VENIX I/O SYSTEM

5.1 INTRODUCTION

This chapter gives an overview of the workings of the VENIX I/O system. It is intended to provide guidance to writers of device driver routines. The chapter describes the environment and nature of device drivers, rather than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The VENIX Time-Sharing System." A more detailed discussion appears in "VENIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the device drivers source, since it is basically an exegesis of that code.

5.2 DEVICE CLASSES

There are two classes of devices: *block* and *character*. The block interface is suitable for devices such as disks, tapes, and DECtape which work, or can work, with addressable 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines, as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as a word (the 'full' device number) with the minor device number in the low byte and the major device number in the high byte. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

5.3 OVERVIEW OF I/O

The purpose of the **open** and **creat** system calls is to set up entries in three separate system tables, as shown in Fig. 1. The first of these is the **u_ofile** table, which is stored in the system's per-process data area **u**. This table is indexed by the file descriptor returned by the **open** or **creat**, and is accessed during a **read**, **write**, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of **open** or **creat**. This table is per-system because the same instance of an open file must be shared among the several processes which can result from **forks** after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 24-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

An entry in the *file* table corresponds precisely to an instance of **open** or **creat**. If the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number from where the entry came.

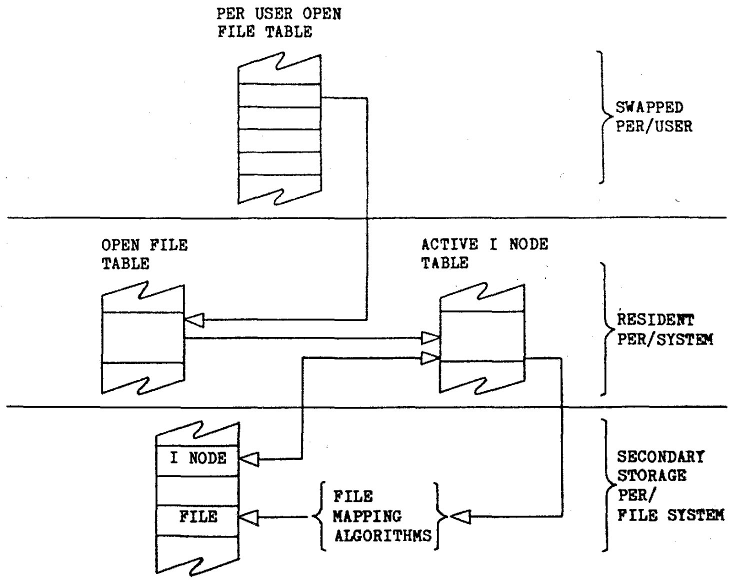

Fig. 1—File system data structure.

During the processing of an **open** or **creat** call for a special file, the system always calls the device's **open** routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the **close** routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. There is one exception to this: when the D_CLOSE (040) bit is on in the minor device number of the special file, **close** is called when any process closes the file it previously opened. Unless this bit is on, however, it is not feasible for a device to maintain, or depend on, a count of its users. Regardless of the D_CLOSE bit, it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a **read** or **write** takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using an indirect block) to a physical block number; a block-type special file need not be mapped. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

[redacted]

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the **ioctl** system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. **open** on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev*; if it should be considered an error, (e.g. **write** on read-only devices) *nodev* is used.

The *open* routine is called each time the file is opened, with the full device number (containing both major and minor indexes) as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The [redacted] is usually called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. However, if the D__CLOSE bit is on in the minor device number of the special file, **close** is called when any of the processes closes the file it opened. Unless the D__CLOSE bit is set, it is not possible for the driver to maintain its own count of its users. [redacted] n the process which caused the **close** routine to be called.

[redacted] The per-user variable *u.u__count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. [redacted] The system may call the routine internally, so [redacted]

The **write** routine should [redacted] from the user's buffer to the device, [redacted] For most drivers, which work one character at a time, the routine [redacted] or an error occurs, when it returns −1. *cpass* takes care of interrogating *u.u__segflg* and updating *u.u__count* and *u.u__base.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine [redacted] which is faster when many characters must be moved. [redacted] Caution: the caller is responsible for making sure the count is not too large and is non-zero. The actual address of [redacted] As an efficiency note, **iomove** is much slower if any of *buffer+offset, count* or *u.u__base* is odd.

The [redacted] is called under conditions similar to **write,** except that *u.u__count* is guaranteed to be non-zero. To [redacted] it takes care of housekeeping like **cpass** and returns −1 as the last character specified by *u.u__count* is returned to the user; before that time, 0 is returned. [redacted]

The 'special functions' routine is invoked by the **ioctl** system call as follows:

[redacted]

where [redacted] to be performed, and [redacted] where the data is to be read/written (as determined by *cmd*). The subroutine

[redacted]

[redacted] and

[redacted]

[redacted] The corresponding subroutines

subyte *(addr,value)*

fubyte *(addr)*

do the same, but for byte values. The two common **ioctl** commands TIOCSETP and TIOGETP (defined in <sgtty.h>) are traditionally used to respectively indicate setting and getting (i.e. returning to user) driver parameters.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices' interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the 'new PS' word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their 'top half' (read/write) and 'bottom half' (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines **getc** and **putc**. [redacted] has the structure

```
struct {
        int     c__cc;   /* character count */
        char    *c__cf;   /* first character */
        char    *c__cl;   /* last character */
[illegible]
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 1200 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call **sleep** *(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority*. The call **wakeup***(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary integer quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area (usually a variable) used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from −127 to 127; a higher numerical value indicates a less-favored scheduling situation. A process sleeping a negative priority cannot be terminated for any reason, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with a negative priority on an event which might never occur. On the other hand, calls to **sleep** with non-negative priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call **sleep** in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area '*u.*' should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a **wakeup,** (for example, a device going on-line, which does not generally cause an interrupt), the call **sleep** *(&lbolt, priority)* may be given. *lbolt* is an external cell whose address is awakened once every second by the clock interrupt routine.

The routines **spl4( ), spl5( ), spl6( ), spl7( ), splx(pri)** are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device; **splx** is a generalized routine which will set the priority level to **pri**. All these routines return the previous processor priority level.

If a device needs to know about real-time intervals, then **timeout** *(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to schedule clock-tick alarms, and to cause the DZ11 input silo to be periodically scanned. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The function **timecancel** *(func, arg)* causes all scheduled timeouts with *func* and *arg* as argument to be cancelled.

5.5 THE BLOCK-DEVICE INTERFACE

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf.* Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw,* *av_back)* which generally maintain a doubly-linked list of blocks which are 'free', that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer. There is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both **bread** and **getblk** return a pointer to a buffer header for the block; the difference is that **bread** is guaranteed to return a buffer actually containing the current data for the block, while **getblk** returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made 'busy', so that other processes referring to it are obliged to wait until it becomes free. **getblk** is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The **breada** routine is used to implement read-ahead. It is logically similar to **bread**, but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the **brelse** routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a **bread**. There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. **bwrite** puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. **bawrite** places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. **bawrite** is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. **bdwrite** is used when there is doubt that the write is needed at the moment. For example, **bdwrite** is called when the last byte of a **write** system call falls short of the end of a block, on the assumption that another **write** will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, **bawrite** is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines **getblk** and **bread** dedicate the given block exclusively to the use of the caller, and make others wait, while one of **brelse, bwrite, bawrite** , or **bdwrite** must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ

This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like **swap** which have a separate argument which indicates read or write.

B_DONE

This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of **getblk** to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR

This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the **b_error** byte of the buffer header may contain an error code if it is non-zero. If **b_error** is 0 the nature of the error is not specified. Actually no driver at present sets **b_error;** the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY

This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When **getblk** (or **bread,** which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_WANTED

This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, **getblk** sets this flag. Conversely, when the block is freed and the busy bit goes down (in **brelse)** a **wakeup** is given for the block header whenever *B_WANTED* is on. This stratagem avoids the overhead of having to call **wakeup** every time a buffer is freed on the chance that someone might want it.

B_ASYNC

This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between **bwrite** and **bawrite** is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that **relse** should be called for the buffer on completion. (This bit should not be confused with asynchronous I/O at the user level via 'raw' I/O.)

B_DELWRI

This bit is set by **bdwrite** before releasing the buffer. When **getblk,** while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

B_AGE

This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_MAP

This bit is set on buffers that have the Unibus map allocated, so that the **iodone** routine knows to deallocate the map.

B_PHYS

This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

5.6 BLOCK DEVICE DRIVERS

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an **open** and a **close** routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a **strategy** routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address (including extended memory bits), the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the **iodone** routine should be called. In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record. [However the mechanism has not been integrated into normal I/O even on magtape and is used only in 'raw' I/O as discussed below.]

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the **swap** routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. However, most disk drivers sort the requests by disk cylinder number to minimize head positioning latency.

A couple of routines are provided which are useful to block device drivers.

iodone (bp)

given a pointer to a buffer header, arranges that it be released or awakened as appropriate to the situation when the driver is finished with the buffer whether normally or after an error. (In the latter case the B_ERROR bit should be set by the driver.)

disksort (&rkutab[unit], bp)

sorts by cylinder number, the buffer pointed to by **bp,** into the queue of outstanding request for a disk driver. Prior to the call, the cylinder number must be put into **b_resid** of the buffer header. **rkutab** is the device table for a particular rk unit (physical drive).

An Example

The RK disk driver is worth studying as a typical example of a block I/O device. Its strategy routine checks to see if the requested block lies beyond the end of the device. If the request is plausible, the buffer is sorted into the outstanding request queue for the drive specified by the minor device number, and if the controller is not busy, **rkstart** is called.

If required, **rkstart** starts a 'seek' on the indicated drive. Otherwise the transfer is started. On a multi-drive system, the rotational position of all drives that have finished seeking is checked, and the transfer is started on the drive with the lowest latency.

When a completion or error interrupt occurs, **rkintr** is called. If an error is indicated, and if the error count has not exceeded 8, the same transaction is reattempted; otherwise the error bit is set. If a transfer is completed or if 8 failing transfers have been issued, the queue is advanced and **rkstart** is called to begin another transaction. If the interrupt is the result of a seek completion, **rkstart** is simply called.

5.7 RAW BLOCK-DEVICE I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device, and providing read and write routines which set up a buffer header with the appropriate information and call the device's strategy routine. If desired, separate open and close routines may be provided, but this is usually unnecessary.

A great deal of work has to be done to generate the 'appropriate information' to put in the argument buffer for the strategy module; the worst part is to map the relocated user addresses to physical addresses. Most of this work is done by

aphysio(strat, dev, rw)

whose arguments are the name of the strategy routine **strat**, the device number **dev**, and a read-write flag **rw** whose value is either **B_READ** or **B_WRITE**. **aphysio** makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it assigns a buffer header for the transaction (or delays until a header is free), fills in the buffer header for the transaction, and calls the strategy routine. Upon return from the strategy call, **aphysio** waits for the transaction to complete, or returns immediately if asynchronous I/O was requested. (Asynchronous I/O is specified by bit 0200 in the minor device number, and bit

0100 further specifies that signal SIGAIO be sent to the user upon I/O completion.)

Note that on some large disk drivers (for example the RM02 driver), the raw I/O feature should be used with caution on inverted partitions. Here the blocks are physically backwards to reduce disk head movement, and thus multi-block transfers will not logically access the correct data if the transfer spirals across cylinders.

Table of Contents

8. SYSTEM MAINTENANCE PROCEDURES

NAME

intro — introduction to device drivers

DESCRIPTION

VENIX uses two types of device interfaces: block and character. The block interface is buffered and handled in chunks of 512 bytes at a time: this includes standard disk I/O and magtape. The character interface is handled on an unbuffered, byte-by-byte basis; all terminals and other serial I/O lines are handled in such a manner. However, disks and magtape can also be treated in a character (read "unbuffered") fashion. This is called "raw" I/O, and is generally faster than regular block I/O.

The most common character device is the terminal, which is explained in detail in **tty**(7). The rest of this introduction is concerned with the ways in which the system and user deal with disks.

In order to successfully utilize the disk resources of a VENIX system it is important to understand the difference between physical and logical representation of disk regions. A single physical disk may be logically divided by the device driver into a number of partitions, each one of which is treated by the software as a separate pseudo-disk. These pseudo-disks may be situated on any part of any physical drive, and some may overlap others. Each pseudo-disk has a unique pseudo-disk (also called "minor device") number, generally between 0 and 7. There is typically one pseudo-disk which represents the entire physical disk volume, while others correspond to portions of a physical volume. The pseudo-disk or "minor device" number assignments for each disk are found in the individual writeups in this section.

Users of VENIX do not customarily deal with these pseudo-disks by their number. Instead, they refer to file names which appear in the root file system in directory /**dev**. The utility **mknod**(1) is used to create entries in /**dev**: it in effect maps a pseudo-disk to a particular /**dev** file entry. This correspondence can be seen in a /**dev** directory listing, where for each file name, the actual pseudo-disk number appears as the second of two numbers in the field where the file sizes are usually placed. (The first number is the "major device" number, indicating which disk controller the pseudo-disk actually resides on.)

For purposes of clarity, the standard names for disks in /**dev** have two parts separated by a dot '.', for example /**dev/rl0.sys**. The rl0 indicates that the device is part of the RL disk unit number 0. Customarily, the first two letters refer to the type of physical drive (in this case, an RL or RL-emulating drive), and the number to its physical drive number (unit

0). The sys indicates that it refers to the pseudo-disk partition used for system programs. Other partitions on the same disk unit would be called **/dev/rl0.usr0**, **/dev/rl0.usr1**, ... referring to various partitions assigned to individual users, **/dev/rl0.tmp** for the temporary areas (used for intermediate files by compilers, editors, and other programs), and **/dev/rl0.all** for the complete physical drive. The latter of course overlaps with all the other partitions. Frequently the .all is simply dropped from the latter and only the preceding part of the name is used: **/dev/rl0**.

Traditionally, file names of the above form are used when doing regular block buffer I/O. If "raw", unbuffered I/O is wanted, the file name is given an additional 'r' before the name; for example, **/dev/rrl0** refers to all of physical drive RL unit 0, to be accessed in raw mode. If asynchronous I/O is used, an 'a' instead of an 'r' precedes the name: **/dev/arl2** refers to all of physical drive RL unit 2, to be accessed asynchronously. (Raw and asynchronous I/O is always done to complete disk units; see **async**(7).)

For the purpose of moving around in the VENIX file system, the general user does not even need to know about the device name associated with his or her individual files, since disks devices are attached or mounted (see **mount**(1)) to directories within the root file system. The VENIX distribution comes with several directories in the root file system — **/u0**, **/u2**, and **/u3** which will be used as the roots of the three user file systems when the system is put into multi-user mode. To keep things simple, the main disk device unit 0 (for example, **/dev/rl0**) is usually mounted on **/u0**, disk device unit 1 on **/u1**, and so on, so it is easy to tell what logical disk all files with a given pathname reside on.

In summary, there are three basic levels on which disks are used. At the highest and most general level (on which general users operate), a file's pathname specifies the location of a file on some file system; the actual storage medium involved is of no importance (although the first component of the name may be some clue to the logical device involved). At the next level (of which the system administrator must have some knowledge), we have device file names in the **/dev** directory, which refer to logical disk partitions. Finally, at the bottom level (known only to the device driver), the mapping between logical partitions and their physical location on disk is made.

BUGS

VENIX should have flag-free disk packs.

NAME

intro — introduction to device drivers

DESCRIPTION

VENIX uses two types of device interfaces: block and character. The block interface is buffered and handled in chunks of 512 bytes at a time: this includes standard disk I/O and magtape. The character interface is handled on an unbuffered, byte-by-byte basis; all terminals and other serial I/O lines are handled in such a manner. However, disks and magtape can also be treated in a character (read "unbuffered") fashion. This is called "raw" I/O, and is generally faster than regular block I/O.

The most common character device is the terminal, which is explained in detail in **tty**(7). The rest of this introduction is concerned with the ways in which the system and user deal with disks.

In order to successfully utilize the disk resources of a VENIX system it is important to understand the difference between physical and logical representation of disk regions. A single physical disk may be logically divided by the device driver into a number of partitions, each one of which is treated by the software as a separate pseudo-disk. These pseudo-disks may be situated on any part of any physical drive, and some may overlap others. Each pseudo-disk has a unique pseudo-disk (also called "minor device") number, generally between 0 and 7. There is typically one pseudo-disk which represents the entire physical disk volume, while others correspond to portions of a physical volume. The pseudo-disk or "minor device" number assignments for each disk are found in the individual writeups in this section.

Users of VENIX do not customarily deal with these pseudo-disks by their number. Instead, they refer to file names which appear in the root file system in directory **/dev**. The utility **mknod**(1) is used to create entries in **/dev**: it in effect maps a pseudo-disk to a particular **/dev** file entry. This correspondence can be seen in a **/dev** directory listing, where for each file name, the actual pseudo-disk number appears as the second of two numbers in the field where the file sizes are usually placed. (The first number is the "major device" number, indicating which disk controller the pseudo-disk actually resides on.)

For purposes of clarity, the standard names for disks in **/dev** have two parts separated by a dot '.', for example **/dev/rl0.sys**. The rl0 indicates that the device is part of the RL disk unit number 0. Customarily, the first two letters refer to the type of physical drive (in this case, an RL or RL-emulating drive), and the number to its physical drive number (unit

0). The sys indicates that it refers to the pseudo-disk partition used for system programs. Other partitions on the same disk unit would be called **/dev/rl0.usr0**, **/dev/rl0.usr1**, ... referring to various partitions assigned to individual users, **/dev/rl0.tmp** for the temporary areas (used for intermediate files by compilers, editors, and other programs), and **/dev/rl0.all** for the complete physical drive. The latter of course overlaps with all the other partitions. Frequently the .all is simply dropped from the latter and only the preceding part of the name is used: **/dev/rl0**.

Traditionally, file names of the above form are used when doing regular block buffer I/O. If "raw", unbuffered I/O is wanted, the file name is given an additional 'r' before the name; for example, **/dev/rrl0** refers to all of physical drive RL unit 0, to be accessed in raw mode. If asynchronous I/O is used, an 'a' instead of an 'r' precedes the name: **/dev/arl2** refers to all of physical drive RL unit 2, to be accessed asynchronously. (Raw and asynchronous I/O is always done to complete disk units; see **async**(7).)

For the purpose of moving around in the VENIX file system, the general user does not even need to know about the device name associated with his or her individual files, since disks devices are attached or mounted (see **mount**(1)) to directories within the root file system. The VENIX distribution comes with several directories in the root file system — **/u0**, **/u2**, and **/u3** which will be used as the roots of the three user file systems when the system is put into multi-user mode. To keep things simple, the main disk device unit 0 (for example, **/dev/rl0)** is usually mounted on **/u0**, disk device unit 1 on **/u1**, and so on, so it is easy to tell what logical disk all files with a given pathname reside on.

In summary, there are three basic levels on which disks are used. At the highest and most general level (on which general users operate), a file's pathname specifies the location of a file on some file system; the actual storage medium involved is of no importance (although the first component of the name may be some clue to the logical device involved). At the next level (of which the system administrator must have some knowledge), we have device file names in the **/dev** directory, which refer to logical disk partitions. Finally, at the bottom level (known only to the device driver), the mapping between logical partitions and their physical location on disk is made.

BUGS

VENIX should have flag-free disk packs.

NAME

ad — analog to digital converter (ADV11-A, DT2762, DT2782, etc.)

DESCRIPTION

ad is a special file which refers to an analog-to-digital converter. A **read** initiates a transfer on channel 0 for the specified number of conversions (byte count divided by 2) with each conversion by default triggered by the "real-time clockc". Each conversion uses one 16-bit word, and the data is right-justified within the word. After the requested number of conversions are completed the **read** returns to the user program; thus, depending on the clock rate and number of conversions, the time for the **read** can be quite long.

If the A/D has direct memory access (DMA) capabilities (for example, Data Translation's DT2782 board), then asynchronous I/O can be used by accessing the special file **aad**.

The previously described default A/D operation can be modified by **ioctl**(2) calls specifying the desired parameters in the following structure:

```
struct  adcntr {
        int     chan;
        int     mode;
};
```

where *chan* is the input channel number (generally 0 through 7), and *mode* can have the following bit combinations:

For an A/D interface without DMA capability (e.g. ADV11-A or DT2762)

GAIN	000	times 1
	004	times 2
	010	times 4
	014	times 8
EXT	020	conversion triggered from external clock
RTC	040	conversion triggered by "real-time clock"

For an A/D interface with DMA capability (e.g. DT2782)

CINC	010	increment channel number after each conversion
BURST	004	convert at maximum rate
RTC	040	conversion triggered by 'real-time clock'

One or the other, but not both of EXT/BURST or RTC must be specified. The following example will select a GAIN of 'times 2' with 'external trigger' and conversions from channel 2 for a non-DMA A/D.

```
    #include <sgtty.h>

    struct adcntr {
            int      chan;
            int      mode;
    } adcntr;


            ...
            afd = open('/dev/ad,'0);
            ...
            adcntr.chan = 2;
            adcntr.mode = 04 | 020;
            ioctl(afd, TIOCSETP, &adcntr);
            ...
            n__bytes = read(afd,...
```

If the special file **adclear** is simply opened, then the current transfer in terminated. This is useful when the A/D is hung because there is no active trigger source.

FILES

/dev/ad	minor device #0
/dev/aad	minor device #8
/dev/adclear	minor device #128

SEE ALSO

ioctl(2), async(7), rtc(7)

LIMITATIONS

The conversion rate is limited to a maximum of 1 to 5 kHz for non-DMA transfers. Asynchronous transfers can also lose 'end of transfer' points (i.e. data between the end of one transfer and the start of the next transfer) for conversion rates greater then 1 to 5 kHz. Both these limitations depend on the processor and other hardware in the configuration, and are primarily due to interrupt latency.

NAME

async — asynchronous raw I/O

DESCRIPTION

Asynchronous I/O (a process continues running while the I/O transfer is taking place) may be used on any Direct Memory Access (DMA) device. This increases a process's I/O throughput and flexibility, at the expense of more programming and buffer overhead. Typically, asynchronous I/O is useful when multiple queuing of buffers is required (the time between transfers is short) or data comes in sporadic bursts and the process must continue processing.

Asynchronous I/O is restricted to raw mode (i.e. character type special files: the VENIX buffering scheme is by-passed) on DMA devices such as disks, many A/D's and D/A's, etc. The minor device number (see **mknod**(1)) of the special file has the following 'magic' values added to get different flavors of asynchronous I/O.

0200 (128)	enable Asynchronous I/O
0100 (64)	send SIGAIO (16) upon I/O completion

For example, suppose the special file '*/dev/rrk2*' exists for raw I/O on drive 2 of a RK05 disk system and it was created by:

```
/etc/mknod /dev/rrk2 c 11 2
```

The asynchronous version would be created by:

```
/etc/mknod /dev/ark2 c 11 130
```

The **aiowait** and the standard **open,seek,read**, etc. system calls are used with asynchronous I/O, with the difference that **read** and **write** system calls immediately return to the caller. The returned transfer count is set as if the transfer actually took place.

It is possible to have the same raw device open for both synchronous (normal) and asynchronous I/O. Keep in mind that raw I/O to disks defeats the VENIX file structure, thus a disk (or disk partition) is generally exclusively used for raw mode I/O.

SEE ALSO

aiowait(2), signal(2), mknod(1), ad(7), da(7), rtc(7)
''VENIX Programming'' for a discussion of asynchronous I/O programming

DIAGNOSTICS

Signal SIGIOT (signal number 6) is sent if there is a device I/O error during asynchronous I/O. It is currently impossible to know anything more about the nature of the error.

BUGS

Since signals are not queued, it is possible to loose a SIGAIO if multiple queuing is used and the queuing rate is faster then the scheduling interval. Thus **aiowait**(2) should be used with fast throughput and multiple queuing.

There is a system dependent maximum number (currently 5) of simultaneously queued asynchronous I/O requests, which if exceeded, causes the caller on the next request to go to sleep until one of the queued asynchronous I/O requests finishes.

NAME

da — digital to analog converter (AAV11-A, DT2766, DT2771, etc.)

DESCRIPTION

da? are special files which refer to the various channels of digital-to-analog converters. A **write** initiates a transfer of the specified number of conversions (byte count divided by 2); each conversion writes out a 16-bit word, and the data must be right-justified within each word.

If the D/A has direct memory access (DMA) capabilities, then asynchronous i/o can be used by accessing the special file **ada?** The DMA version of the D/A interface also has several options which can be invoked by using **ioctl**(2) calls specifying the desired parameters in the following structure:

```
struct  dacntr {
        char    mode;
        char    data;
};
```

Where *data* is 4 bits of data (in the low 4 bits) output to TTL connections on the interface and *mode* can be −1 for single channel output operation, or 0 for dual channel operation appropiate for driving a storage scope. The following example selects the dual channel operation.

```
#include <sgtty.h>
struct dacntr {
        char    mode;
        char    data;
} dacntr;
        ...
        dfd = open('/dev/da0,'1);
        ...
        dacntr.data = 0;
        dacntr.mode = 0;
        ioctl(dfd, TIOCSETP, &dacntr);
        ...
        n_bytes = write(dfd,...
```

FILES

/dev/da0	minor device # 0	
/dev/da1	minor device # 1	
.	.	
.	.	
/dev/ada0	minor device # 128	(asynchronous)
/dev/ada1	minor device # 129	(asynchronous)
.	.	
.	.	

SEE ALSO

ioctl(2), async(7)

NAME

dh — DH-11 asynchronous multiplexer

DESCRIPTION

Each line attached to the DH-11 asynchronous multiplexer behaves as described in **tty**(7). Input and output for each line may be independently set to run at any of 16 speeds; see **ioctl**(2) for the encoding.

FILES

/dev/tty
/dev/tty?? ...

SEE ALSO

tty(7), ioctl(2)

NAME

dl — KL-11 or DL-11 asynchronous interface

DESCRIPTION

The discussion of typewriter I/O given in **tty**(7) applies to these devices. Since they run at a constant speed, attempts to change the speed via **ioctl**(2) are ignored.

The on-line console typewriter is interfaced using a DL-11 or KL-11. During a system boot, VENIX will come up single-user with I/O on the console.

FILES

/dev/console	minor device #0
/dev/tty	
/dev/tty?? ...	

SEE ALSO

tty(7), init(8)

BUGS

The DL tends to lose characters due to interrupt latency when operating at high baud rates or during a period of heavy system usage. For this reason, an asynchronous interface with an input silo, such as the DZ, is recommended for modems.

Modem control for the DL is not implemented.

NAME

dz — DZ-11 asynchronous multiplexer

DESCRIPTION

Each line attached to the DZ-11 asynchronous multiplexer behaves as described in **tty**(7). Input and output for each line may be set to run at any of 16 speeds; see **ioctl**(2) for the encoding.

FILES

/dev/tty?? ...
/dev/tty
/dev/tty

SEE ALSO

tty(7), ioctl(2)

NAME

ht — RH-11/TU-16 magtape interface

DESCRIPTION

The files **mt0 ... mt7** refer to the DEC RH/TM/TU16 magtape. The 020 bit (decimal 16) on in the minor device number selects single density. If the 040 bit (decimal 32) is on in the minor device number the tape will not be rewound when closed, and the head will be left positioned between the two tapemarks. (The files **nmt0 ... nmt7** have this bit on.) If the file was open for writing, two end-of-files are written when it is closed.

A standard tape consists of a series of 512 byte records terminated by an end-of-file. To the extent possible, the system allows the tape to be treated (though perhaps inefficiently) like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The **mt** files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named **rmt0 ... rmt7.** Each **read** or **write** call reads or writes the next record on the tape. In the write case the record is the same length as the buffer given. During read, the record size is passed back as the number of bytes read, provided it is no greater then the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file. Asynchronous I/O can be done in the usual way.

FILES

/dev/mt?	(1600 bpi tape)
/dev/mt?s	(800 bpi tape)
/dev/rmt?, /dev/rmt?s	(1600, 800 bpi raw tape)
/dev/nmt?, /dev/nmt?s	(1600, 800 bpi no-rewind-on-close tape)
/dev/nrmt?, /dev/nrmt?s	(1600, 800 bpi raw, no-rewind-on-close tape)
/dev/amt?, /dev/amt?s	(1600, 800 bpi asynchronous tape)

SEE ALSO

tar(1), tp(1)

DIAGNOSTICS

On unrecoverable errors, the following message is printed on the console:

Error on dev X/Y
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is control-status register 1; **B** is the error register.

BUGS

If any non-data error is encountered, the handler refuses to do anything more until the device is closed. In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

Taking a drive off line, or running off the end of tape while writing, have been known to hang the system.

NAME

ib11 — IEEE488 controller driver

DESCRIPTION

This driver handles input and output on the DEC IBV11 bus controller board in a multi-user compatible fashion. Not all features of the IEEE488 standard bus are implemented, most significantly, service requests. Those available include reading and writing to primary bus addresses, go-to-local, selected-device-clear, group-execute-trigger, and local-lock-out.

All bus protocol and handshaking are transparent to the user (ie. ATN, MLA, UNT and so forth). Thus the user may not need to become too familiar with the standard. In practice, some less sophisticated commercial devices have their own special quirks which may require special attention.

A separate special file is made for each device which is to use the bus. The minor device number is the same as the bus address of the peripheral, typically beginning with 0 and increasing. Low numbered devices have higher priority. However, to assure general usefulness, the driver will change devices only at the end of messages.

Note that a disconnected device will not cause an error message on opening, but on the first attempted transaction. Certain types of peripherials, notably Phillips devices, will object to the sending of null bytes, and other means of determining if they are indeed present. An IBC (interface-bus-clear) signal is sent when the first device is opened. A device may be opened only once.

I/O may then be done via standard read and write calls. The STDIO library may be used as well with many devices, but proper attention should be paid to buffering. Talker and listener addressing is done automatically.

The **ioctl(2)** calls use the following structure defined in **ibv.h**:

```
struct sgibv {
        unsigned short ib__flags;       /* see below */
        char ib__trm2, ib__trm1;        /* terminator character #2
                                        or 1 char terminator */
        char ib__trm1, ib__trm1;        /* terminator character #1 or 0
                                        if only a single terminator */
        short ib__timo;                 /* timeout in ticks */
        };
```

The IIOCSETP command, defined in **ibv.h** is used to set (write) these values. Usage is:

```
ioctl (fd, IIOCSETP, ps);
```

where **ps** is a pointer to an sgibv structure. The flag bits encoded in **ib__flags** are:

```
#define  C__ONETERM  0000002   /* one character terminator seq*/
#define  C__ERROR    0010000   /* there has been an error since
                                  last read or write began */
#define  C__TERMCHK  0002000   /* look for terminators */
```

The default terminator is a <cr><lf> combination. If the terminator is found on a read, input is assumed to be finished. On a write, terminators merely define where a higher priority device may slip in and where the EOI line should be set. Reads will also terminate if the device sets the EOI line. Terminators may be turned off by clearing the C__TERMCHK bit.

The timeout feature is optional. If a request is pending for longer than the specified number of clock ticks, an error return occurs. The default timeout delay is 600 ticks (10 seconds). A timeout of 0 disables this feature.

The IIOCCMD command, defined in **ibv.h**, is used to send a command onto the bus. The following commands are guaranteed to behave properly:

```
#define C__GTL 001                    /* ieee commands */
#define C__SDC 004
#define C__GET 010
#define C__LLO 021
#define C__DCL 024
```

Usage is:

```
ioctl (fd, IIOCCMD, p);
```

where **p** is a pointer to an integer containing a command. If an error or service request interrupt occurs, the devices **ibs** and **ibd** registers will be printed out on the console.

FILES

/dev/device-name — minor device number = bus address of device
/usr/sys/dev/ibv.h

NOTES

Service requests and polling are not available. Nonstandard devices (CAMAC interfaces) which use the UNT and UNL signals as terminators may need spurious reads or writes.

The driver uses a full 8 bit interface. A signal during a transaction will cause no problem worse than a spurious error message on the next transaction to that device.

NAME

lp — line printer

DESCRIPTION

lp provides the interface to any of the standard DEC line printers. When it is opened or closed a suitable number of page ejects is generated. Bytes written are printed.

There are numerous parameters in the driver which can be modified for local printer variations. The driver needs to be recompiled into the system to effect these changes.

FILES

/dev/lp

SEE ALSO

lp(1)

BUGS

Many systems have the local printer on an asynchronous interface. To avoid confusing **lp**(1), this interface line, though through a DL or asynchronous line, is called **/dev/lp**. Thus **/dev/lp** is really a generic name for the local printer.

NAME

mem, kmem — primary memory

DESCRIPTION

mem is a special file that is an image of the primary memory of the computer. It may be used, for example, to examine, and even patch the system. **kmem** is the same as **mem** except that kernel virtual memory rather than physical memory is accessed.

Byte addresses are interpreted as memory address. Reading non-existent locations finds −1 at those places.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

On PDP11's, the I/O page begins at location 0160000 of **kmem** and per-process data for the current process begins at 0140000.

FILES

/dev/mem, /dev/kmem

BUGS

On PDP11's, memory files are accessed one byte at a time, an inappropriate method for some device registers.

NAME

null — data sink

DESCRIPTION

Data written to a null special file is callously discarded. Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

pio— parallel digital input/output (DEC DRV11 or DR11-C)

DESCRIPTION

pio is the special file which refers to the parallel digital I/O interface. A **write** transfers the requested number of words (byte count divided by 2) to the output register. Likewise, a **read** transfers the requested number of words from the input register.

The "control and status" register can be written or read by using **ioctl**(2) calls with a TIOCSETP (write) or TIOCGETP (read) request, and a pointer to the following structure.

```
struct   piocntr {
         int      csr;
         int      n_intra;
         int      n_intrb;
};
```

where **csr** has the following bit definitions:

CSR0	0001	control bit on connector 2
CSR1	0002	control bit on connector 1
IEA	0100	interrupt enable request A
IEB	0040	interrupt enable request B
REQA	0000200	request A
REQB	0100000	request B

N_intra and **n_intrb** contain the number of interrupts since the previous **ioctl** call (with a TIOCGETP command).

FILES

/dev/pio

SEE ALSO

ioctl(2), tty(7)

BUGS

The interrupt handling is rather crude, but difficult to implement in a general way. Interested users are refered to the driver source, which they may modify to suit their specific needs.

NAME

rk05 — RK05 disk

DESCRIPTION

rk? refers to an entire disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871. Minor device numbers are drive numbers on one controller.

The following names are used for disk partitions. Each partition is an entire unit.

minor no.	starting block	length in blocks	usage	suggested name
0	0	4872	user	/dev/rk0
1	0	4872	user	/dev/rk1
2	0	4872	user	/dev/rk2
3	0	4872	user	/dev/rk3

The **rk** files discussed above access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RK files begin with **rrk** and end with a number which selects the same disk as the corresponding **rk** file. Additionally, raw RK files can be accessed asynchronously (**async(7)**) by using the name **ark** ending with a number which selects the same disk as the corresponding **rk** file. In raw I/O the buffer must begin on a word boundary.

FILES

/dev/rk?, /dev/rrk?

NOTES

Error-free disks are strongly recommended.

DIAGNOSTICS

On unrecoverable errors, the following message is printed on the console:

```
Error on dev X/Y
bnN erA B
```

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the error register; **B** is the drive status register.

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 512-byte block boundaries, and **write** scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, **read,write** and **lseek**(2) should always deal in 512-byte multiples.

NAME

rk06 — RK06/07 disk

DESCRIPTION

rk0.sys ... rk0.usr refer to sections of RK disk drive 0. The files **rk1.usr** refer to drive 1 (if present) and so on. This allows a large disk to be broken up into more manageable pieces. Each RK06 drive contains 27126 512-byte blocks and each RK07 contains 53790 512-byte blocks.

The origin and size of the disk partitions are as follows. The table is extended for 2 RK06/07 disks. (In the following table, the alternative partition sizes for RK06 and RK07 drives are given respectively in pairs.)

minor no.	starting block	length in blocks	usage	suggested name
0	0	7854	root(6816) swap(1038)	/dev/rk0.sys
1	7854	2442	tmp	/dev/rk0.tmp
2	10296	16830 (rk06) 53790 (rk07)	user	/dev/rk0.usr
3	0	27126 (rk06) 53790 (rk07)	complete drive 0	/dev/rk0.all
4	0	27126 (rk06) 53790 (rk07)	user, unit 1	/dev/rk1.usr
5		unassigned		

The **rk** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RK files begin with **rrk** and end with a name which selects the same disk section as the corresponding **rk** file. Additionally, raw RK files can be accessed asynchronously (**async**(7)) by using the name **ark** followed by a name which selects the same disk section as the

corresponding **rk** file.
In raw I/O the buffer must begin on a word boundary.

NOTES

Error-free disks are strongly recommended.

FILES

/dev/rk*, /dev/rrk*, /dev/ark*

DIAGNOSTICS

On recovered ECC errors, the following is printed on the console:

N ECC on dev *X*/*Y*

X and *Y* are the major and minor device numbers. Error is in block number *N*.

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is control-status register 1; **B** is the error register

BUGS

In raw I/O **read** and **write(2)** truncate file offsets to 512-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

NAME

rl — RL01/RL02 disk

DESCRIPTION

rl0.sys ... rl0.usr refer to sections of RL disk drive 0. The files **rl1.usr0** , **rl1.usr1** refer to drive 1 (if present) and so on. This allows a large disk to be broken up into more manageable pieces. Each RL02 drive contains 20480 512-byte blocks and each RL01 contains 10240 512-byte blocks.

The origin and size of the disk partitions are as follows. The table is extended for 2 RL02 disks.

minor no.	starting block	length in blocks	usage	suggested name
0	0	20480	complete drive 0	/dev/rl0.all
1	0	7840	root(6816) swap(1024)	/dev/rl0.sys
2	7840	2400	tmp	/dev/rl0.tmp
3	10240	10240 (rl02) – (rl01)	user (rl02 only)	/dev/rl0.usr
4	0	20480 (rl02) 10240 (rl01)	complete drive 1	/dev/rl1.usr
5	0	10240	half drive one	/dev/rl1.usr0
6	10240	10240 (rl02) – (rl01)	half drive one (rl02 only)	/dev/rl1.usr1
7		unassigned		

The **rl** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RL files begin with **rrl** and end with a name which selects the same disk section as the corresponding **rl** file. Additionally, raw RL files can be accessed asynchronously (**async**(7)) by using the name **arl** followed by a name which selects the same disk section as the corresponding **rl** file.

In raw I/O the buffer must begin on a word boundary.

FILES

/dev/rl*, /dev/rrl*, /dev/arl*

NOTES

Error-free disks are strongly recommended.

DIAGNOSTICS

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the control-status register; **B** is the word-count register

BUGS

In raw I/O **read** and **write(2)** truncate file offsets to 256-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

NAME

rm02 — RM02/RM03 disk

DESCRIPTION

rm0.usr0 ... rm0.usr2 refer to sections of RM disk drive 0. The files **rm1.usr0 ... rm1.usr2** refer to drive 1, etc. This allows a large disk to broken up into more manageable pieces. Each RM02 disk drive contains 131,680 512-byte blocks (67 mbytes).

The origin and size of the pseudo-disks are as follows:

minor no.	starting block	length in blocks	usage	suggested name
0	0	160	for boot block	/dev/rm0.boot
1	160	50880	user (inverted)	/dev/rm0.usr0
2	51040	7840	root(6816) swap(1024)	/dev/rm0.sys
3	58880	2400	tmp	/dev/rm0.tmp
4	61280	50880	user	/dev/rm0.usr1
5	112160	19520	small user	/dev/rm0.usr2
6		unassigned		
7		unassigned		

The **rm** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RM files begin with **rrm** and end with a name which selects the same disk section as the corresponding **rm** file. Additionally, raw RM file can be accessed asynchronously (see **async**(7)) by using the name **arm** followed by a name which selects the same disk section as the corresponding **rm** file.

In raw I/O the buffer must begin on a word boundary.
Head positioning latency (seek time) often can be reduced by a judicious layout of the file systems on a disk. The RM offers some illustrations. First of all, larger physical disks are often partitioned into smaller logical sections; in the RM's case this is a necessity because of the 32 mbyte maximum file system size. Given this, then the fact that on a file system the i-nodes are at the beginning with data tending to extend toward the end (the last used area), makes it clear that two files systems back-to-back (the physically first file system being inverted) will not have an unused area to be skipped over between file systems, as two forward file systems do. Since the root file system is most frequently referenced, it is best to sandwich it between two back-to-back file systems. Finally, an infrequently used file system should be put at the physical end of the disk.

FILES

/dev/rm*, /dev/rrm*, /dev/arm*

DIAGNOSTICS

On recovered ECC errors, the following is printed on the console:

N ECC on dev *X*/*Y*

X and *Y* are the major and minor device numbers. Error is in block number *N*.

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is control-status register 2; **B** is error register 1.

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 512-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

NOTES

Error-free disks are strongly recommended.
Raw I/O of more then single block transfers should not be used on an inverted file system because a logical forward sequence of blocks is physically reversed.

NAME

rm05 — RM05 disk

DESCRIPTION

rm0.usr0 ... rm0.usr7 refer to sections of RM disk drive 0. The files **rm1.usr0 ... rm1.usr7** refer to drive 1, etc. This allows a large disk to broken up into more manageable pieces. Each RM05 disk drive contains 500,384 512-byte blocks (256 mbytes).

The origin and size of the pseudo-disks are as follows:

minor no.	starting block	length in blocks	usage	suggested name
0	0	608	for boot block	/dev/rm0.boot
1	6	65056	user (inverted)	/dev/rm0.usr0
2	65664	9728	root(6816) swap(2912)	/dev/rm0.sys
3	75392	3648	tmp	/dev/rm0.tmp
4	79040	65056	user	/dev/rm0.usr1
5	144096	65056	user	/dev/rm0.usr2
6	209152	65056	user	/dev/rm0.usr3
7	274208	65056	user	/dev/rm0.usr4
8	339264	65056	user	/dev/rm0.usr5
9	404320	65056	user	/dev/rm0.usr6
10	469376	31008	small user	/dev/rm0.usr7
11		unassigned		

The **rm** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation therefor raw I/O is considerably more efficient when many words are transmitted. The names of the raw RM files begin with **rrm** and end with a name which selects the same disk section as the corresponding **rm** file. Additionally, raw RM file can be accessed asynchronously (see **async**(7)) by using the name **arm** followed by a name which selects the same disk section as the corresponding **rm** file.

In raw I/O the buffer must begin on a word boundary.

FILES

/dev/rm*, /dev/rrm*, /dev/arm*

DIAGNOSTICS

On recovered ECC errors, the following is printed on the console:

N ECC on dev *X*/*Y*

X and *Y* are the major and minor device numbers. Error is in block number *N*.

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is control-status register 2; **B** is error register 1.

NOTES

Error-free disks are strongly recommended.

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 512-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

Raw I/O of more then single block transfers should not be used on an inverted file system because a logical forward sequence of blocks is physically reversed.

NAME

shutdown — exit from multi-user to single-user

SYNOPSIS

/etc/shutdown

DESCRIPTION

shutdown checks that all users (except super-users) have logged off and there are no background processes running, thus preventing accidental termination of users. (If desired, **shutdown** will send warning messages to all users every 30 seconds before beginning the shutdown.) Then all mounted file systems are unmounted (except the root file system) by examining **/etc/mtab** after which the signal SIGTERM is sent to the **/etc/init** process (pid 1), informing **/etc/init** to exit from multi-user.

shutdown is usually not executed as a command, but rather as a substitute shell for the 'shutdown' login. **login shutdown** is used to run it.

SEE ALSO

kill(2), init(8)

FILES

/venix	system namelist
/etc/init	
/etc/mtab	

NAME

update — periodically update the super block

SYNOPSIS

/etc/update

DESCRIPTION

update is a program that executes the **sync**(2) primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file **/etc/rc**.

SEE ALSO

sync(2), sync(1), init(8)

BUGS

With **update** running, if the CPU is halted just as the **sync** is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. **shutdown**(8) prevents this problem.

NAME

rp02 — RP02/RP03 disk

DESCRIPTION

rp0.usr0 ... rp0.usr2 refer to sections of RP disk drive 0. The files **rp1.usr0 ... rp1.usr2** refer to drive 1, etc. This allows a large disk to broken up into more manageable pieces. Each RP02 disk drive contains 40,600 512-byte blocks (20 mbytes), and each RP03 disk drive contains 81,200 512-byte blocks (40 mbyte).

The origin and size of the pseudo-disks are as follows, extended for three disk units:

minor no.	starting block	length in blocks	usage	suggested name
0	0	8000	sys(6816) swap(1184)	/dev/rp0.sys
1	8000	2400	tmp	/dev/rp0.tmp
2	10400	30200	user	/dev/rp0.usr0
3	40600	40600 (rp03) – (rp02)	user	/dev/rp0.usr1 (rp03 only)
4	0	40600	user, start drive 1	/dev/rp1.usr0
5	40600	40600 (rp03) – (rp02)	user	/dev/rp1.usr1 (rp03 only)
6	40600	0	user, start drive 2	/dev/rp2.usr0
7	40600	40600 (rp03) – (rp02)	user	/dev/rp2.usr1 (rp03 only)

The **rp** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single

The **rp** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with **rrp** and end with a name which selects the same disk section as the corresponding **rp** file (e.g. **/dev/rrp0.usr1**). Additionally, raw RP files can be accessed asynchronously (see **async**(7)) by using the name **arp** followed by a name which selects the same disk section as the corresponding **rp** file (e.g. **/dev/arp0.usr1**).

In raw I/O the buffer must begin on a word boundary.

DIAGNOSTICS

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the control-status register; **B** is the error register

FILES

/dev/rp*, /dev/rrp*, /dev/arp*

NOTES

Error-free disks are strongly recommended.

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 512-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

Raw I/O of more then single block transfers should not be used on an inverted file system because a logical forward sequence of blocks is physically reversed.

NAME

rp — RP04/RP05 and RP06 disk

DESCRIPTION

rp0.usr0 ... rp0.usr2 refer to sections of RP disk drive 0. The files **rp1.usr0 ... rp1.usr2** refer to drive 1, etc. This allows a large disk to broken up into more manageable pieces. Each RP04/05 disk drive contains 171,798 512-byte blocks (88 mbytes), and each RP06 disk drive contains 340,670 512-byte blocks (174 mbyte).

The origin and size of the pseudo-disks are as follows:

minor no.	starting block	length in blocks	usage	suggested name
0	0	418	for boot block	
1	418	65208	user (inverted)	/dev/rp0.usr0
2	65626	8778	root(6816) swap(1962)	/dev/rp0.sys
3	74404	32186	tmp	/dev/rp0.tmp
4	106590	65208	user (end of rp04/05)	/dev/rp0.usr1
5	171798	65208	user (rp06 only)	/dev/rp0.usr2
6	237006	65208	user (rp06 only)	/dev/rp0.usr3
7	302214	38456	small user (rp06 only)	/dev/rp0.usr4

The **rp** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with **rrp** and end with a name which selects the same disk section as the corresponding **rp** file. Additionally, raw RP files can be accessed asynchronously (see **async**(7)) by using the name **arp** followed by a name which selects the same disk section as the corresponding **rp** file.

In raw I/O the buffer must begin on a word boundary.

FILES

/dev/rp*, /dev/rrp*, /dev/arp*

DIAGNOSTICS

On recovered ECC errors, the following is printed on the console:

N ECC on dev *X*/*Y*

X and *Y* are the major and minor device numbers. Error is in block number *N*.

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is control-status register 2; **B** is error register 1.

NOTES

Error-free disks are strongly recommended.

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 512-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

Raw I/O of more then single block transfers should not be used on an inverted file system because a logical forward sequence of blocks is physically reversed.

NAME

rq — Micro/PDP-11 winchester/floppy RQDX controller

DESCRIPTION

The **rq** driver handles both the Micro/PDP-11 winchester and floppy controllers. Special files **w0.sys, w0.tmp**, and **w0.usr** refer to winchester partitions; **f0** and **f1** refer to the floppy drives.

The origin and size of the floppy and hard disk partitions are as follows:

minor no.	starting block	length in blocks	usage	suggested name
0	0	790	floppy drive 0	/dev/f0
1	0	790	floppy drive 1	/dev/f1
2	–	–	unused	–
3	–	–	unused	–
4	0	7840	root(6816) swap(1024)	/dev/w0.sys
5	7840	2400	temporary files	/dev/w0.tmp
6	10240	10240	user	/dev/w0.usr0
7	20480	40000	user (optional)	/dev/w0.usr1

The special files discussed above access the disk via the system's normal 'block' buffering mechanism and may be read without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RQ files begin with **rw0** or **rf** and end with a name which selects the same disk section as the corresponding block file. Additionally, raw files can be accessed asynchronously (**async**(7)) by using the name **arw** or **arf** followed by a name which selects the same disk section as the corresponding block file.

In raw I/O the buffer must begin on a word boundary.

FILES

/dev/w0*, /dev/f?, /dev/rw0*, /dev/rf?, /dev/arw*, /dev/arf?

NOTES

Error-free disks are strongly recommended.

DIAGNOSTICS

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the receive status register; **B** is the physical block number

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 256-byte block boundaries, and **write** scribbles on the tail of incomplete blocks.

NAME

rtc — real time clock (DEC KWV11)

DESCRIPTION

rtc is the special file which refers to the real time clock interface. This interface is used to time intervals or count events and then generate an interrupt and/or trigger an a/d conversion, etc. The "control and status" register can be written or read by using **ioctl**(2) calls with a TIOCSETP (write) or TIOCGETP (read) request, and a pointer to the following structure.

```
struct  rtc {
        int     csr;
        int     preset;
};
```

where **csr** can have the following values:

GO	0001	start counter action
MODE	0006	
MODE0	0000	mode 0 operation
MODE1	0002	mode 1 operation
MODE2	0004	mode 2 operation
MODE3	0006	mode 3 operation
RATE	0070	
STOP	0000	stop the clock
E6HZ	0010	1 MHz clock rate
E5HZ	0020	100 kHz clock rate
E4HZ	0030	10 kHz clock rate
E3HZ	0040	1 kHz clock rate
E2HZ	0050	100 Hz clock rate
ST1	0060	Schmitt trigger 1
LINE	0070	line frequency rate
INTOV	0100	interrupt on overflow flag
OVFLO	0200	overflow flag
MAINT	0007400	maintenance bits
FOR	0010000	flag overrun
ST2GO	0020000	Schmitt trigger 2 enable
INTST2	0040000	interrupt enable Schmitt 2
ST2FLG	0100000	Schmitt trigger 2 flag

and **preset** is the value read from or to be written to the buffer preset register. **N_intr** is the number of interrupts since the previous **ioctl** call.

FILES

/dev/rtc

SEE ALSO

ioctl(2), tty(7)

BUGS

The interrupt handling is rather crude, but difficult to implement in a general way at the user level. Interested users are referred to the driver source, which they may modify to suit their specific needs.

NAME

rx — RX01/RX02/RX03 floppy disk

DESCRIPTION

Rx? and **rx?s** refers to an entire disk as a single sequentially-addressed file. Its logical 256-word blocks are numbered 0 to 493 for single density disks **rx?s** and 0 to 987 for double density disks **rx?**. Minor device numbers 0 and 1 encode drives 0 and 1 at double density, while 2 and 3 (i.e. adding in 2) specifies single density on drives 0 and 1. If the floppy drive is double sided (RX03), then the maximum number block is 987 and 1976 for single and double density disks respectively.

The **rx** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerable more efficient when many words are transmitted. The names of the raw RX files begin with **rrx** and end with a number which selects the same disk as the corresponding **rx** file. Additionally, raw RX files can be accessed asynchronously (see **async**(7)) by using the name **arx** followed by a number which selects the same disk as the corresponding **rx** file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 128 bytes (single density) or 256 bytes (double density): a disk block. Likewise **seek** calls should specify a multiple of 128 or 256 bytes.

The standard DEC interleave factor of 2 and skew factor of 6 are used.

FILES

/dev/rx?, /dev/rrx?, /dev/arx?, /dev/rx?s

DIAGNOSTICS

On unrecoverable errors, the following is printed on the console:

> Error on dev *X*/*Y*
> bn*N* er*A* **B**

X and *Y* are the major and minor device numbers.
Error is in block number *N*;
A is the data buffer register; **B** is the control status register

BUGS

In raw I/O **read** and **write**(2) truncate file offsets to 128 (single density) of 256 (double density) –byte block boundaries, and **write** scribbles on

the tail of incomplete blocks.
Only single density is supportable on RX01's. Also, since the RX01 does not use DMA transfers, the raw and asynchronous modes cannot be supported.

NAME

sms — Scientific Micro System FWT floppy/winchester disks

DESCRIPTION

sms.sys ... sms.usr1 refer to sections of Winchester disk drive 0. The files **sms1.all** refers to drive 1 (if present) and so on. **rx0** refers to the RX02-compatible double-sided double-density floppy drive 0, and **rx1** refers to floppy drive 1 (if present). An 's' following the floppy name **(rx0s)** refers to the RX01 single-density version of that floppy; an 'i' following the name **(rx0i)** refers to the IBM format double-density version.

The total size of the Winchester disks range from 8.9 to 35.6 mbytes; the floppy disks are 1 mbyte in length in double density, and

The origin and size of the various disks and partitions are as follows.

minor no.	starting block	length in blocks	usage	suggested name
0	0	1976	DS floppy 0	/dev/rx0
1	0	1976	DS floppy 1	/dev/rx1
2	0	988	SD floppy 0	/dev/rx0s
3	0	988	SD floppy 1	/dev/rx1s
4	0	7840	winchester 0, system (6816 sys; 1024 swap)	/dev/sms.sys
5	7840	2400	winchester 0, tmp	/dev/sms.tmp
6	10240	10240	winchester 0, user	/dev/sms.usr0
7	20480	?	rest of win 0	/dev/sms.usr1
8	0	2448	IBM DD floppy 0	/dev/rx0i
9	0	2448	IBM DD floppy 1	/dev/rx1i

(The partition with length '?' varies in length depending on the physical size of the disk.)

The **rx** and **sms** files discussed above access the disk via the system's normal buffering mechanism and may be read without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation, therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw special files begin with **rrx** and **rsms** and end with a name which selects the same disk section as the corresponding **rx** and **sms** file.

Additionally, raw special files can be accessed asynchronously (**async**(7)) by using the name **arx** and **asms** followed by a name which selects the same disk section as the corresponding **rx** and **sms** file. In raw I/O the buffer must begin on a word boundary.

ioctl calls may be done (see **ioctl**(2)) on raw versions of floppies to format diskettes. While formatting is going on, all activity on the Winchester drive or other floppy drive is suspended. The **ioctl** command used to format is the ASCII captal letter 'X' shifted left into the high byte of the command word, as in

```
ioctl(fd,'X'<<8,0);
```

d is the file descriptor returned by an **open** call to one of the raw floppy devices.

FILES

/dev/rx*, /dev/rrx*, /dev/arx*, /dev/sms*, /dev/rsms*, /dev/asms*

DIAGNOSTICS

On unrecoverable errors, the following is printed on the console:

Error on dev *X*/*Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the data buffer register; **B** is the control status register.

BUGS

In raw I/O **read**(2) and **write**(2) truncate file offsets to 512-byte block boundaries on the Winchester and 256-byte block boundaries on the floppy disks, and **write** scribbles on the tail of incomplete blocks.

WARNING

Double-sided floppy diskettes are read and written in a manner incompatible with DEC standards.

NAME

tc — TC-11/TU56 DECtape

DESCRIPTION

The files **tap0 ... tap7** refer to the TC-11/TU56 DECtape drives 0 to 7. The 256-word blocks on a standard DECtape are numbered 0 to 577.

FILES

/dev/tap?

SEE ALSO

tp(1)

DIAGNOSTICS

On unrecoverable errors, the following is printed on the console:

```
        Error on dev X/Y
        bnN erA 0
```

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the control-status register.

NAME

tm — TM11/TU10 or TS11 magtape interface

DESCRIPTION

The files **mt0, ..., mt7** refer to the DEC TU10/TM11 or TS11 magtape. If the 040 bit (decimal 32) is on in the minor device number the tape will not be rewound when closed, and the head will be left positioned between the two tapemarks. (The files **nmt0 ... nmt7** have this bit on.) If the file was open for writing, two end-of-files are written when it is closed.

A standard tape consists of a series of 512 byte records terminated by an end-of-file. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The **mt** files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named **rmt0, ..., rmt7.** Each **read** or **write**(2) call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?, /dev/rmt?, /dev/nmt?, /dev/nrmt?, /dev/amt?

SEE ALSO

tar(1), tp(1)

DIAGNOSTICS

On unrecoverable errors on TU10/TM11 devices, the following is printed on the console:

Error on dev *X/Y*
bn*N* er*A* **B**

X and *Y* are the major and minor device numbers. Error is in block number *N*; *A* is the control-status register; **B** is the error register.

For TS11 errors, see driver source.

BUGS

If any non-data error is encountered, the driver refuses to do anything more until the device is closed.

In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

NAME

tty — general terminal interface

DESCRIPTION

This section describes both a particular special file, and the general nature of the terminal interface.

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

As for terminals in general: all of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of the interface.

When a terminal file is opened, the systems causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by **init**(8) and become a user's input and output file. The very first terminal file open in a process becomes the **control terminal** for that process. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a **fork**, even if the control terminal is closed. The set of processes that thus share a terminal is called a **process group**; all members of a process group receive certain signals together (see ^C below and **kill**(2)).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even when output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet be read some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information. (Once the user has typed the newline character, the line is buffered internally and can be read in any size chunks). There are special modes, discussed below, that permit the program to read each character as typed without waiting for a full line.

During input, erase and kill processing is normally done. By default, the 'delete' character erases the last character typed, except it will not erase beyond the beginning of a line or EOT. If the terminal is a CRT, then an attempt is made to remove the preceding character (by a backspace, space, and backspace sequence); otherwise a '<' is echoed. By default, the ' ^U' character (control u; typed by depressing the 'control' key and typing 'u') is set as the 'kill' character: it kills the entire line up to the point where it was typed, but not beyond the beginning of the line or EOT. Both the delete and kill characters may be changed by the user.

When desired, all upper-case letters may be mapped into the corresponding lower-case letter on input, and everything echoed to the terminal in upper-case only. Real upper-case letters may be generated by preceding it by an 'escape', which echos as '$'. In addition, the following escape sequences can be generated on output and accepted on input:

for	use
`	$'
|	$!
~	$^
{	$(
}	$)

See the LCASE flag below.

Certain ASCII control characters have special meaning. These characters are not passed to a reading program except in raw mode when they lose their special meaning.

^D (EOT) may be used to generate an end-of-file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication.

^C is not passed to a program but generates an **interrupt** signal which is sent to all processes with the associated control terminal. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See **signal**(2). All typed characters not yet read are killed.

^Z generates the **quit** signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated.

^S delays all printing on the terminal until anything is typed in; it is not echoed. Usually a ^Q is used to start output again.

^Q restarts printing after a ^S, or, in SCROLL mode, after screen output stops; it also is not echoed.

^R reviews (re-echos) all typed characters that have not yet been read by any program.

^E kills all typed in characters that have not yet been read by any program.

BREAK
Identical to '^C', but works even in RAW mode and does not kill all unread characters.

When the carrier signal from a dataset drops (usually because the user has hung up his remote terminal) a **hangup** signal is sent to all processes with the terminal as their control terminal. Unless other arrangements have been made, this signal causes the process to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file on their input can terminate appropriately when hung up on.

Terminal I/O is buffered: when characters are written to the terminal, they are actually sent as soon as previously-written characters have finished being typed; input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly then they can be sent out, it will be suspended when its output queue exceeds some limit. When the queue has drained to some threshold the program is resumed.

Several **ioctl**(2) calls apply to terminals. Most of them use the following structure, defined in <**sgtty.h**>.

```
struct sgttyb {
        charsg__ispeed;
        charsg__ospeed;
        charsg__erase;
        charsg__kill;
        intsg__flags;
};
```

The **sg__ispeed** and **sg__ospeed** fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface (and closely to the DZ-11). If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are defined in <**sgtty.h**>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

Currently, only 300 and 1200 baud are really supported on dial-up lines. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supported; full duplex 212 datasets work fine. Speed cannot be changed on terminals attached to a DL-11 or KL-11; the VENIX console, whose special use is described in **boot**(8), is one such terminal.

The **sg__erase** and **sg__kill** fields of the argument structure specify the erase and kill characters respectively. (Defaults are 'delete' and ' ^U'.)

The **sg_flags** field in the argument structure contains several bits that determine the system's treatment of the terminal:

CRT	0100000	Terminal is a CRT
SCROLL	0040000	Output stops automatically every 20 lines
XTABS	0006000	Expand tabs to spaces on output
RAW	0000040	Raw mode: 8 bit interface (turns off CRT, XTABS, CRMOD, LCASE and CBREAK)
EVENP	0000200	Enables even parity
ODDP	0000100	Enables odd parity
CRMOD	0000020	Map CR into LF; echo LF as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper to lower case (Escapes work)
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The undefined bits are used on some UNIX systems to specify delays on certain characters and parity; these delays and parity are currently ignored under VENIX.

Several **ioctl** calls have the form:

```
#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;
```

The applicable codes are:

TIOCGETP Fetch the parameters associated with the terminal, and store in the pointed-to structure.

TIOCSETP Set the parameters according to the pointed-to structure. The interface delays until output is quiescent, then throws any unread characters, before changing the modes.

TIOCSETN Set the parameters but do not delay or flush input.

TIOCEXCL Set 'exclusive-use' mode: no further opens are permitted until the file has been closed.

TIOCNXCL Turn off 'exclusive-use' mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH
All characters waiting on input or output are flushed.

TIOCQCNT Returns the count of characters currently typed in but not yet read in **sg_ispeed** (0 to 255) and the count of characters on the output queue, which is probably rapidly changing, in **sg_ospeed** (0 to about 100).

FILES

/dev/tty	the user terminal
/dev/tty*	all terminals on the system
/dev/console	the console terminal

SEE ALSO

stty(1), signal(2), ioctl(2), getty(8)
"VENIX Programming," for an example of using **ioctl.**

NOTES

The following differences are noted between the VENIX tty handler and that in some other versions of UNIX:

1. Reading in RAW mode returns the number of characters asked for, not just one as in other versions;
2. The SCROLL mode and TIOCQCNT command (return character count in buffer) are not supported by some other versions;
3. delay mode for output of certain characters is not supported by VENIX;
4. the interrupt character (^C), delete character (DEL) and kill character (^U) are different than in some other versions. (The delete and kill characters, however, can be changed by ioctl calls or the stty command.)

BUGS

Half-duplex terminals are not supported.
The deleting of characters on a CRT cannot be exact, due to the interspersing of input and output. Furthermore, 'delete' assumes that one character's position is to be removed, inappropriate for tabs and control characters. If there is confusion as to exactly what has been deleted, the re-echo (^R) feature should clear it up.

Handling of CTRL-S and CTRL-Q as well as SCROLL mode is often useless with terminals which automatically send CTRL-S and CTRL-Q codes

NAME

vp — Versatec or Benson/Varian printer/plotter

DESCRIPTION

The device node **lp** refers to a Versatec C-PDP(DMA) or Benson/Varian controller. Ordinarily bytes written on it are interpreted as ASCII characters and printed. Only some of the ASCII control characters are interpreted:

NL	Performs the usual new-line (line-feed) function, i.e. spaces up the paper and resets to the left margin. It is ignored however following a CR which ends a non-empty line.
CR	is ignored if the current line is empty but is otherwise like NL.
FF	resets to the left margin and then to the top of the next page.
EOT	resets to the left margin, advances 8 inches, and then performs a FF.

The **ioctl**(2) system call may be used to change the mode of the device. The format of the call is

```
ioctl(fd,request,&vmode)
```

fd is the file descriptor for the open device, and request is the value ('v'<<8 | 0) for a 'get mode', and ('v'<<8 | 1) for a 'set mode'. The mode is written to or read from the integer **vmode**. The bits of **vmode** mean:

0400	Enter simultaneous print/plot mode.
0200	Enter plot mode.
0100	Enter print mode (default on open).
0040	Send remote terminate.
0020	Send remote form-feed.
0010	Send remote EOF.
0004	Send remote clear.
0002	Send remote reset.

On a 'get mode', only the first three conditions (print, plot, or print/ plot) are indicated. Notice that the mode bits above are not encoded, so that it is required that exactly one be set. When the Versatec is opened, a reset and clear are automatically done, and it is placed in print mode. On close an EOF is done.

In plot mode each byte is interpreted as 8 bits of which the high-order is plotted to the left; a '1' leaves a visible dot. A full line of dots is produced by 264 bytes; lines are terminated only by count or by a remote terminate function. There are 200 dots per inch both vertically and horizontally.

When simultaneous print/plot mode is entered exactly one line of character should be written, terminated by NL, CR or the remote terminate function. Then the device enters plot mode and at least 20 lines of plotting bytes should be sent. As the line of characters (which is 20 dots high) is printed, the plotting bytes overlay the characters. Notice that it is impossible to print characters on baselines that differ by fewer than 20 dot-lines.

In print mode lines may be terminated either with an appropriate ASCII character or by using the remote terminate function.

DIAGNOSTICS

"Versatec needs attention!" if errors occur.

FILES

/dev/lp

SEE ALSO

lp (1)

NAME

boot — startup procedures

DESCRIPTION

The bootstrap must reside in the otherwise unused block zero of the boot device. It can be read in and started by the standard ROM programs, or if necessary by keying in a small startup routine. This bootstrap is capable of loading type 407 executable files (not shared, not separate I&D). The user is prompted on the system console with a '&'. The name of the program to be loaded, usually **venix**, is typed in followed by a **CR**. Typing errors cannot be corrected. The named file is retrieved from the file system that starts on block 0 of the device booted by the ROM. If the file cannot be found, the prompt is reissued.

After **venix** has started running and sized memory, it prints a copyright message and the amount of memory available for users in kilobyte units. The root and pipe file systems (see **/usr/sys/conf/c.c** for definitions) are then mounted. **/etc/init** starts running, which in turn opens **/dev/console** and starts **/bin/sh**. The shell prompts the user on the system console; the system is now running single-user. After doing any file system checks and setting the date, the user can bring the system up in multi-user mode by typing a ^D (CTRL-D).

FILES

/venix	
/usr/mdec/*	backup bootstraps

SEE ALSO

init(8), a.out(4)

BUGS

If the kernel symbol table is too large, it may overwrite a portion of the bootstrap and prevent proper loading. Stripping the kernel with **strip**(1) will prevent this; an unstripped copy should be maintained also, however, for the benefit of **ps** and **shutdown**.

NAME

cron — clock daemon

SYNOPSIS

/etc/cron [min]

DESCRIPTION

cron executes commands at specified dates and times according to the instructions in the file **/usr/lib/crontab.** Since **cron** never exits, it should only be executed once. This is best done by running **cron** from the initialization process through the file **/etc/rc.**

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1 = monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. (It is not permissable to combine a list and a range in one field.) The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character.

Crontab is examined by **cron** every *min* minutes, or 1 if no number is specified on the command line. It sets its effective ID to 1 ('sys' user) while running.

FILES

/usr/lib/crontab

NAME

getty — set typewriter mode

SYNOPSIS

/etc/getty [char]

DESCRIPTION

getty is invoked by **init**(8) immediately after a typewriter is opened following a dial-up. It reads the user's login name and calls **login**(1) with the name as an argument. While reading the name **getty** attempts to adapt the system to the speed and type of terminal being used.

init calls **getty** with a single character argument taken from the **/etc/ttys** file entry for the terminal line. This argument determines a sequence of line speeds through which **getty** cycles, and also the 'login:' greeting message, which can contain character sequences to put various kinds of terminals in useful states.

The user's name is terminated by a newline or carriage-return character. In the second case CRMOD mode is set (see **ioctl**(2)), so that future carriage-returns will be treated as newlines.

The name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

If the terminal's 'break' key is depressed, **getty** cycles to the next speed appropriate to the type of line and prints the greeting message again.

Finally, login is called with the user's name as an argument. See **ttys**(4) for a list of terminal options in the **/etc/ttys** file.

SEE ALSO

init(8), login(1), ioctl(2), ttys(4)

have been deactivated in the file, and login processes are created for terminals which have been made active.

FILES

/dev/tty?, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

SEE ALSO

login(1), kill(1), sh(1), shutdown(8), ttys(4), getty(8)

NAME

init, rc — process control initialization

SYNOPSIS

/etc/init
/etc/rc

DESCRIPTION

init is invoked as the last step of the boot procedure. Generally its role is to create a process for each typewriter on which a user may log in.

When **init** first is executed the console typewriter **/dev/console** is opened for reading and writing and the shell is invoked immediately. This feature is used to bring up a single-user system. If the shell terminates, **init** comes up multi-user and the process described below is started.

When **init** comes up multiuser, it invokes a shell, with input taken from the file **/etc/rc**. This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then **init** reads the file /**etc/ttys** and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input, output and error files. Opening the typewriter will usually involve a delay, since the open is not completed until someone is dialed up and carrier established on the channel. Then /**etc/getty** is called with argument as specified by the last character of the **ttys** file line. **getty** reads the user's name and invokes **login**(1) to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of **init**, which has been waiting for such an event, wakes up and removes the appropriate entry from the file **utmp**, which records current users, and makes an entry in **/usr/adm/wtmp**, which maintains a history of logins and logouts. Then the appropriate typewriter is reopened and **getty** is reinvoked.

init catches the hangup signal SIGHUP and interprets it to mean that the system should be brought from multi user to single user. The **shutdown login** normally accomplishes this; alternatively, 'kill -1 1' can be used to send the hangup signal.

Sending an interrupt signal SIGINT to **init** (as by 'kill -2 1') causes **init** to reexamine the /**etc/ttys** file; processes are killed for terminals which