CONVEX SPU System Manager's Guide

First Edition

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
P.O. Box 833851
Richardson, TX 75083-3851
(214) 497-4000

CONVEX
SPU
System Manager's Guide

Order No. DSW-022

First Edition
November 1991

CONVEX Press
Richardson, Texas
United States of America

CONVEX
SPU
System Manager's Guide

Order No. DSW-022

Printed in the United States of America

Revision information for

CONVEX
SPU
System Manager's Guide

Edition	Document No.	Description
First	710-018230-001	Initial release, released with ConvexOS V10.0, November 1991. Contains information previously found in *Managing ConvexOS: Operations Guide.*

Contents

Figures

Tables

Using this guide

Purpose and audience

This guide describes to the system manager:

- Hardware and software components that make up the SPU
- Purpose of the SPU
- How to use the SPU to manage the C1, C200 Series, C3200 Series, and C3400 Series CONVEX systems. References to the C3200 Series in this book include the C200 Series.

If you do not have one of these machines, you should use the *CONVEX C3800 Series SPU System Manager's Guide* to manage the SPU on your system.

This guide applies equally to machines running ConvexOS and ConvexOS/Secure, although, in the text, the operating system is referred to only as ConvexOS.

Organization

If you are running ConvexOS, this guide should be used in conjunction with *Managing ConvexOS: Configuration Guide* and *Managing ConvexOS: Operations Guide*. If you are running ConvexOS/Secure, this guide should be used in conjunction with *Managing ConvexOS/Secure: Configuration Guide* and *Managing ConvexOS/Secure: Operations Guide*.

This guide describes SPU-related tasks. *Managing ConvexOS: Configuration Guide* and *Managing ConvexOS/Secure: Configuration Guide*d describes tasks required to configure ConvexOS. *Managing ConvexOS: Operations Guide* and *Managing ConvexOS/Secure: Operations Guide* describes tasks related to monitoring and maintaining the system.

Notational conventions

This section discusses notational conventions used in this book.

General conventions

The following conventions are used in this guide:

- *Italics*
  - Designate user-supplied variables in a command-line example
  - Indicate document titles
- `Constant-width font` designates input that must be typed exactly as it appears and output displayed on the terminal screen. This includes:
  - Command names and options
  - Directives, program statements, display examples, printout examples, and error messages returned.
- **`Bold constant-width font`** identifies user input in examples.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-x** indicates that you must press and hold down the **CTRL** key and then press the **x** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

Command syntax

In order to use the commands in this document, you must understand the conventions used when describing command syntax. Consider this example:

`command` *input_file* `[`*input_file* `...] {a|b} [`*output_file*`]`

① ② ③ ④ ⑤

1. `Constant-width font` indicates that you must type the characters exactly as they appear (uppercase and lowercase are identical).
2. *Italics* indicate a variable that must be supplied by the user. In this case, the user must supply the name of an *input_file*.
3. Square brackets [ ] indicate optional data. Horizontal ellipsis (...) shows repetition of the preceding item(s). In this case, the user can optionally specify more than one *input_file* on the command line.
4. Curly brackets { } indicate a choice. The choices available are shown inside the curly brackets and separated by the pipe (|) sign. In this case, the user can enter either `a` or `b`.
5. [*output_file*] indicates the user can optionally specify an output file name with the command.

Associated documents

Using this software may require information not specific to the tasks described in this document.

For more information on the ConvexOS operating system, you can order these books from CONVEX Computer Corporation:

- *Managing ConvexOS: Configuration Guide* (DSW-030). Describes tasks required to configure ConvexOS.
- *Managing ConvexOS: Operations Guide* (DSW-031). Describes tasks required to monitor and maintain the system.
- *Managing ConvexOS/Secure: Configuration Guide* (DSW-577). Describes tasks required to configure ConvexOS/Secure.
- *Managing ConvexOS/Secure: Operations Guide* (DSW-581). Describes tasks required to monitor and maintain the system.

Ordering documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

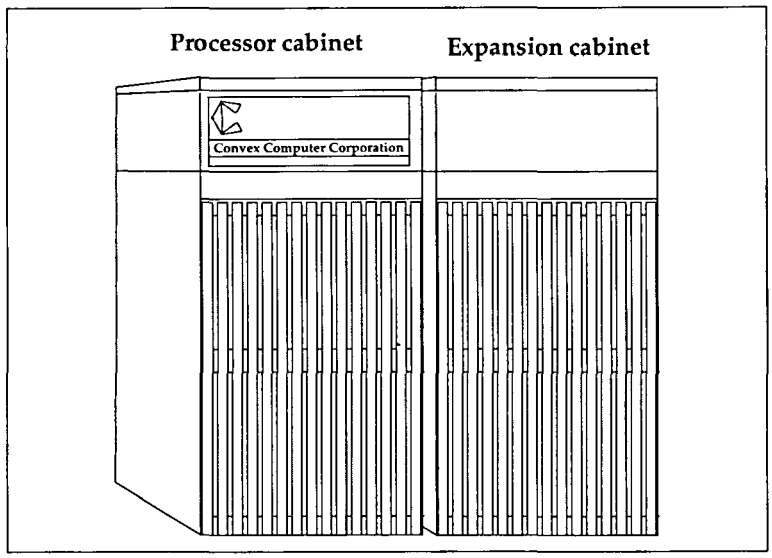
- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384
- Outside continental U.S., contact local CONVEX office.

SPU hardware and software

1

The basic CONVEX supercomputer consists of one or two processor cabinets and one or more expansion cabinets (see Figure 1).

Figure 1 Processor and expansion cabinet



The processor cabinet contains:

- Processor boards
- Memory boards
- Channel Control Units (CCUs)
- Power supplies and power supply control panel (AC power-controller panel)
- Front control panel
- Multibus or VMEbus chassis
- System monitor board or system control monitor
- SPU hardware

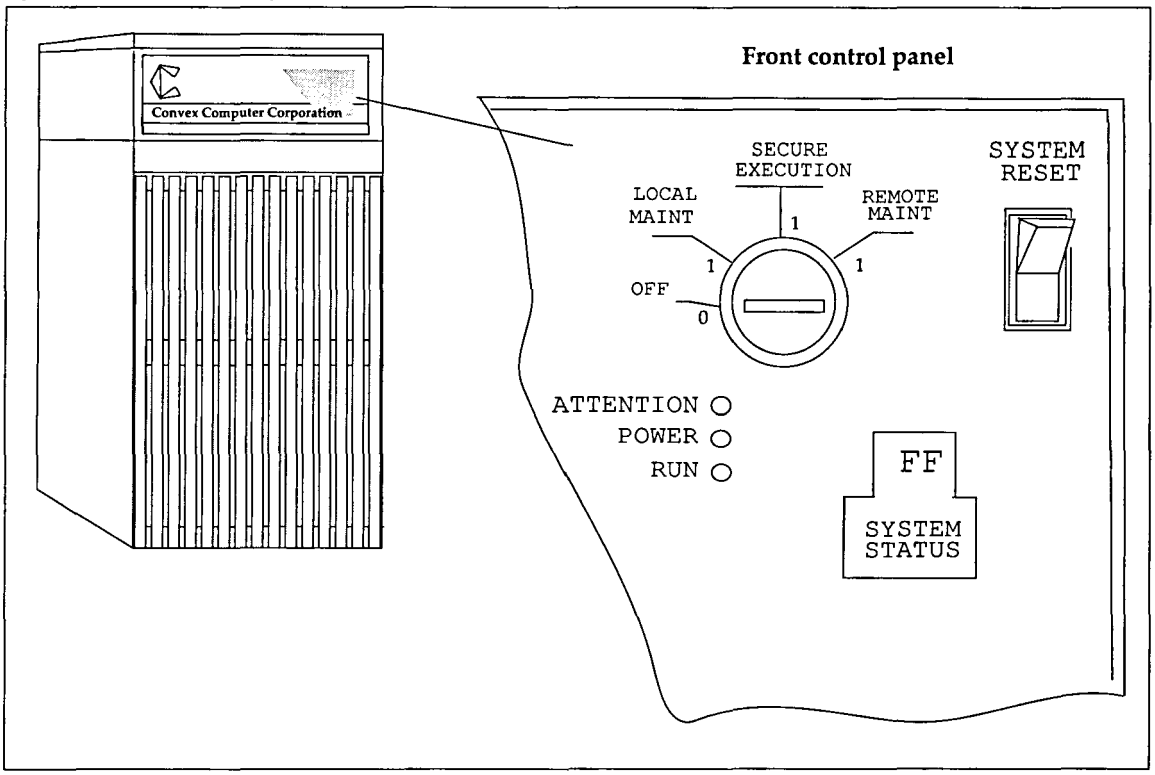
The expansion cabinets provide the physical means to increase peripheral and I/O capacity. Additional expansion cabinets can be added to the system as required. Expansion cabinets can contain:

- Single tape drive
- One or more disk drives
- Modem
- Peripheral controllers
- RS-232-C interface panel

Front control panel

The front control panel is essential to SPU operations. The computing environment you enter when you power on the system depends on the keyswitch setting located on the front control panel. See Figure 2 for an illustration of the keyswitch and its location on the front control panel.

Figure 2 Location of keyswitch



The front control panel is located at the top of the leftmost bay inside the front door of the cabinet containing the SPU. The keyswitch has four positions:

OFF	Allows no access to the SPU from the system console.
LOCAL MAINT	Allows local access to the SPU from the system console.
SECURE EXECUTION	Prevents direct access to the SPU from the system console and disables the SYSTEM RESET button.
REMOTE MAINT	Allows remote access to the SPU through the SPU modem port. Used primarily to run remote diagnostics.

Caution

If you are running ConvexOS/Secure, networking is not supported in the evaluated configuration. It is recommended that you do not provide networking access in a trusted environment.

SPU hardware

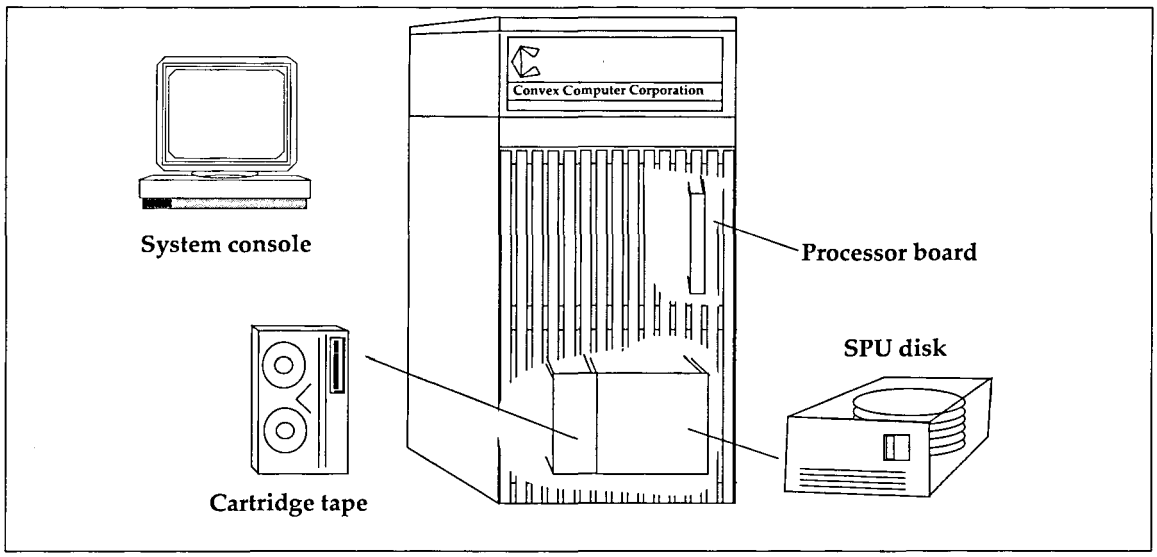
The SPU is a completely independent computer system with a single-board microprocessor, on-board local memory, and peripheral interfaces, such as boot devices and consoles. The SPU is completely separate from CONVEX CPU system resources.

The SPU is used to:

- Initialize the CPUs on the CONVEX machine
- Boot ConvexOS
- Collect error information, since it continues to run after the processor is halted due to an error in one of its functions
- Run diagnostics on CONVEX hardware components

The hardware that makes up the SPU is located in the processor cabinet(s). Figure 3 shows the location of the SPU hardware components in a single processor cabinet configuration. Refer to the hardware manual for your machine for specific information about your configuration.

Figure 3 Location of SPU hardware components



The SPU components are:

- Service processor unit (SPU) board
- Cartridge tape drive available to SPU OS

  The SPU tape drive is used for loading SPU software distributions (such as SPU OS) onto the SPU disk. It is also possible to boot the SPU from the SPU tape drive, assuming a bootable tape is available.

- Hard disk available to SPU OS

  On the C1 Series machine, the SPU disk is a 5.25 inch 20-Mbyte hard disk. On the C3200 machine, which incudes the C200 Series, the SPU disk is a 5.25 inch 146-Mbyte hard disk with an embedded disk controller. On the C3400 Series machine, the SPU disk is a 5.25 inch 535-mbyte hard disk with an embedded controller.

  The SPU disk is the default SPU boot device. If the Removable Disk System (RDS) is installed, the removable SPU hard disk is located in the expansion cabinet. The SPU disk contains:
  - SPU OS
  - vmunix (kernel image)
  - I/O processor operating systems
  - System boot and error utilities
  - All offline diagnostics
  - Error logs
  - Files containing processor configuration and performance information

- System console

  The system console is used to issue operating system commands to the system. It communicates with the SPU and with the ConvexOS through the SPU. The system console also displays system-generated messages to inform and assist system management. It is typically a VT100 compatible CRT terminal. The system console can either be a local console or a remote console.

  The local console is connected to the SPU via an RS-232 port set to 9600 baud. This port is enabled only when the front control panel keyswitch is set to the LOCAL MAINT position. The local console also has a console printer attached to it through the auxiliary port.

Caution

Restrict physical access to the system console to those users authorized to access all information processed by the system.

The remote console port is enabled when the front control panel keyswitch is set to the REMOTE MAINT position. In REMOTE MAINT, no keyboard input is accepted from the local console. Input is submitted remotely through a modem and a remote connection. However, all commands submitted remotely appear on the local console screen.

Caution

If you are running ConvexOS/Secure, networking is not supported in the evaluated configuration. It is recommended that you do not provide networking access in a trusted environment.

SPU software

The SPU disk contains all or portions of the three operating systems required to run the CONVEX machine:

- SPU OS

  SPU OS operates the SPU and controls processes that are requested to run in the SPU. These processes include tasks such as finding a particular file on the SPU disk or tape drive, scanning processor boards, and handling messages between the CCUs. SPU OS also schedules all on-going jobs running on the SPU, which are swapped in and out as needed.

- ConvexOS

  The kernel image (vmunix) is contained on the SPU disk so that it can be loaded into system memory at boot time. This code is sufficient to configure the CPU so that it can find its own root partition on the system disk and complete the boot process.

- I/O processors

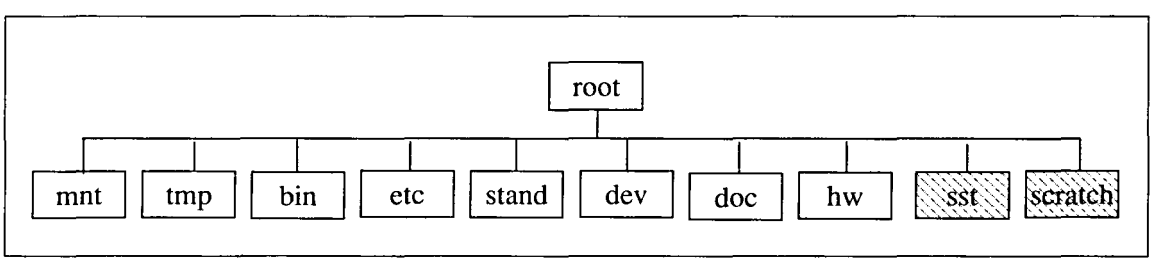
  The operating system for I/O processors is kept on the SPU disk and loaded into main memory at system boot time. At the time of I/O processor initialization, a copy of the I/O processor operating system in main memory is loaded into CCU local memory and executed.

The SPU disk also contains all utilities required to:

- Initialize and boot SPU OS and ConvexOS
- Maintain system operation
- Create and retain records of system performance

These functions are performed by SPU-based programs that execute under SPU OS. Figure 4 illustrates the directory structure for SPU OS. The clear boxes are the directories common to all series of machines, the shaded boxes are directories unique to the C3200 Series, and the striped boxes are directories unique to the C3400 Series.

Figure 4 Directory structure for SPU OS



The purpose of each directory is listed below:

mnt	Contains the ConvexOS kernel image (vmunix) and other files needed for the boot process, all diagnostics, and all diagnostic configuration files.
tmp	Contains temporary files created by system processes. This directory is not backed up.
bin	Contains SPU OS utility programs, such as `more` and `pwd`.
etc	Contains system management utilities, such as `fsck` and `backup`.
stand	Contains stand-alone test programs, such as spu2000.
dev	Contains special files for devices, such as tape drives or disk drives.
doc	Contains release notices for diagnostics and the diagnostic database.
hw	Contains the product engineering tests.
sst	Contains system scan tests and data files. This directory is not backed up.
scratch	Used for scratch purposes. This directory is not backed up.

See Appendix A for more information on the commands and files found in these directories.

Important boot files on the SPU

The programs and most of the files required to boot ConvexOS are located in the /mnt/os directory on the SPU. Files important to the boot process are described in the following sections.

bootcmd and bootcmd.local files

When the system is booted (always from the SPU), ConvexOS is loaded from the /mnt/os/vmunix file, operating system parameters are read from the /mnt/os/bootcmd file, and if it exists, additional parameters are read from the /mnt/os/bootcmd.local file.

The /mnt/os/bootcmd file is provided with your ConvexOS release tape and contains information about how to boot your system, where the root partition resides, and commands that specify system parameters. You should not alter this file, as it is subject to change with each release of ConvexOS. Instead, you should use the /mnt/os/bootcmd.local file to set boot-time parameters specific to your site.

The /mnt/os/bootcmd.local file is not part of the ConvexOS release tape. You must create it to change default values for kernel boot-time parameters. Commands in the /mnt/os/bootcmd.local file take precedence over those in /mnt/os/bootcmd, allowing you to customize the way your system boots. See the "Customizing Kernel Boot-Time Parameters" chapter in *Managing ConvexOS: Configuration Guide* for details on how to modify the /mnt/os/bootcmd.local file.

/ioconfig file

The /ioconfig file is also located on the SPU and contains a description of all Channel Control Units (CCUs), interfaces, controller boards, and peripheral devices for your system. The boot process reads this file to determine what devices are present. When adding a device to your system, you must integrate the new device into the system by modifying the /ioconfig file to include the description for the new device. See the "Adding Devices" chapter in *Managing ConvexOS: Configuration Guide.*

Once the operating system is up and running, the `init` program is started. This program is the parent process for all other system processes and always has a Process Identification (PID) of 1. `init` executes /etc/.initrc then the /etc/rc startup script, which starts system processes and prepares the system for multiuser operation.

Computing environments

When you power up the system, you can power up to one of three computing environments:

- Soft front panel
- SPU OS
- ConvexOS

Information about each of these environments is provided in the following sections.

Soft front panel environment

The soft front panel is the first level in the environment hierarchy and runs on the Service Processor Unit (SPU). You must use the system console to issue commands to the soft front panel. The soft front panel is used primarily to modify soft front panel options, to select system boot options, and to run the self test for the SPU. See Chapter 3, "Setting soft front panel options," for details on setting soft front panel options.

The soft front panel is an interactive program stored in Erasable Programmable Read-Only Memory (EPROM) on the SPU circuit board. The SPU also stores the soft front panel and boot options in nonvolatile EPROM memory, so the settings are preserved even when the system is powered off.

SPU OS environment

SPU OS is the second level in the hierarchy and runs on the SPU. You must use the system console to issue commands to SPU OS. SPU OS functions are:

- Controlling CONVEX diagnostic software
- Coordinating error logging
- Booting the CPU

SPU OS is designed for smaller applications and is, therefore, not a virtual-memory system. Its primary command interpreter is the Bourne shell, although the C shell is available.

ConvexOS environment

ConvexOS is the third level in the hierarchy and is a demand-paged, virtual-memory operating system derived from Berkeley 4.2 BSD. You can use the system console or any workstation in multiuser mode to issue commands to ConvexOS. ConvexOS is the normal computing environment and consists of:

- Command interpreters (shells)
- Utilities
- A file system
- A kernel that manages all system resources, including some interrupts, memory management, process control, and I/O

Like all operating systems, ConvexOS acts primarily as a coordinator and scheduler of system resources. ConvexOS supports two standard command interpreters (also called shells) that enable users to issue commands to the system:

- Bourne shell (`sh`)
- C shell (`csh`)

Two modes of operation exist within the ConvexOS computing environment: single-user mode and multiuser mode.

Single-user mode

Only one user can use the system in single-user mode: the superuser. In single-user mode, the superuser must use the system console to access the system. By default, only directories in the root partition are available, although you can manually mount other file systems. The superuser, typically the system manager, uses single user mode to:

- Run maintenance software
- Perform file system checks
- Mount and unmount file systems

Multiuser mode

Multiple users can log in and use system resources in multiuser mode, including the superuser or system manager. Among other things, the system manager uses multiuser mode to:

- Monitor user file space
- Check on the number of users currently logged on
- Monitor the amount of user processing time

Environment prompts

Each computing environment has its own system prompt. You can always tell which environment you are in by the system prompt displayed on the terminal. Table 1 lists system prompts for each environment.

Table 1 Prompts for computing environments

Environment	Prompt
Soft front panel	(fp)>
SPU OS	(SPU)>
ConvexOS	
login	login:
single-user mode	#
multiuser mode	# (as root)
multiuser mode (csh)	% (as user)
multiuser mode (sh)	$ (as user)

The soft front panel, SPU OS, and single-user mode prompts appear only on the SPU console. All other prompts can appear on either the SPU console or a user terminal. Prompts for both single-user mode and multiuser mode as root are the pound sign (#). To distinguish one mode from the other, enter the following command at the # system prompt:

```
who
```

If no one, not even root, is logged in, you are in single-user mode.

Setting soft front panel options

3

Commands available on the soft front panel allow examination and modification of soft front panel options. These options control how the system acts when booted. For example, whether the system boots to multiuser mode, diagnostic mode, or an alternate mode.

When the soft front panel is started, the current settings of the soft front panel options are displayed as shown in Figure 5.

Figure 5 Soft front panel default settings

```
123456789ABCDEF
CONVEX Front Panel- Version: 3.32 / CPU Class: 7 SN 32514
SPU type = SP5                    Processor = 68000
mode of operation = normal-os     boot-device = disk
location-of-bootstrap = default   power-up-reboot = enable
automatic-reboot = enable         spu-selftest = enable
test-flags = normal               remote-port-bps = 1200
SCSI-power-up-delay = 0xA         user-flags = 0x0
 (fp)>
```

Each hexadecimal digit (123456789ABCDEF) in the first line of Figure 5 represents successive phases in the SPU self-test procedure. If the self-test encounters an error, the last hexadecimal digit displayed on the system console indicates the self-test phase where the error was detected. For example, if the SPU self-test failed at phase six of the test, the system console displays 123456.

Entering `help` at the `(fp)>` prompt displays a list of the soft front panel commands.

The system allows you to enter the first letter or letters of the command that uniquely identifies the command string. For example, entering

```
s m=n
```

is the same as entering

```
set mode-of-operation=normal-os
```

Table 2 lists the commands available from the soft front panel. The minimal letters required to execute the command are shown in boldface letters.

Table 2 Soft front panel commands

Comnd	Description
`boot`	Loads and invokes the disk or tape bootstrap. The source of the boot program is determined by the `boot-device` soft front panel option.
`display`	Displays all soft front panel settings.
`help`	Displays a one-page listing of available soft front panel commands.
`preset`	Provides a method to set all of the soft front panel options to a predefined state. This command is useful when setting all options or if you wish to return to a known soft front panel state. The command format is: `preset` *option* where *option* can be one of the following. Each of the options represent a default configuration. `standard` Sets the following configuration: mode-of-operation normal-os boot-device disk location-of-bootstrap default power up reboot enabled automatic-reboot enabled spu-selftest enabled test-flags normal remote-port-bps 1200 SCSI-power-up-delay 0xA user-flags 0x0 `alternate` Sets the following configuration: mode-of-operation alternate-os boot-device disk location-of-bootstrap default power up reboot enabled automatic-reboot enabled spu-selftest enabled test-flags normal remote-port-bps 1200 SCSI-power-up-delay 0xA user-flags 0x0

Table 2 Soft front panel commands (continued)

Comnd	Description
	diagnostic Sets the following configuration: mode-of-operation diagnostic boot-device disk location-of-bootstrap default power up reboot disabled automatic-reboot disabled spu-selftest enabled test-flags normal remote-port-bps UNCHANGED SCSI-power-up-delay 0xA user-flags 0x0
	install Sets the following configuration: mode-of-operation other boot-device tape location-of-bootstrap default power up reboot disabled automatic-reboot disabled spu-selftest enabled test-flags normal remote-port-bps 1200 SCSI-power-up-delay 0xA user-flags 0x0
set	Modifies the soft front panel options. The command format is: **s**et *option=value* where *option* can be one of the following: **m**ode of operation Controls how the boot command behaves when SPU OS or ConvexOS is booted. *value* can be one of the following: **n** (normal-os) When the soft front panel boot command executes with the keyswitch in the LOCAL MAINT or REMOTE MAINT positions, SPU OS is booted and then ConvexOS is booted. **a** (alternate-os) System prompts the system manager to select either single-user or multiuser mode before ConvexOS is booted.

Table 2 Soft front panel commands (continued)

Comnd	Description	
		d (diagnostic) System boots SPU OS or an alternate diagnostic mode. **o** (other) ConvexOS is not booted.
	boot-device	Selects the location of the boot device. *value* can be one of the following: **d** (disk) Selects the SPU disk as the boot device. **t** (tape) Selects a cartridge tape as the boot device. Mount a copy of the SPU OS boot image tape in the SPU tape drive before booting the system, if tape is selected.
	location-of-bootstrap	Selects one of four redundant copies of the boot program used by the system. In effect, this option provides access to three backup copies of the boot program if errors are discovered in the default boot program. Do not change the value of this field without instructions from the CONVEX Technical Assistance Center. *value* can be one of the following: **d** (default) Selects the default copy of the boot program. # Selects one of three backup copies of the boot program. # can be 1, 2, or 3. This option is reserved for diagnostic use.
	power-up-reboot	Controls whether or not the system executes the boot procedure without entering the soft front panel after reset from a power up condition. This option is only applicable if the keyswitch is on the SECURE EXECUTION position. The automatic-reboot option must be enabled for this to work. *value* can be one of the following: **e** (enabled) Reboots to multiuser mode. **d** (disabled) Reboots to soft front panel.

Table 2 Soft front panel commands (continued)

Comnd	Description	
	automatic-reboot	Specifies whether or not the system automatically executes the boot procedure without entering the soft front panel after a reset condition. This option is applicable only if the keyswitch is in the SECURE EXECUTION position. *value* can be one of the following: **e** (enabled) Enables the reboot command. **d** (disabled) Disables the reboot command.
	spu-self-test	Specifies whether or not the SPU performs a self-test when booting. The self-test is an initial hardware check that verifies that all boot devices are installed properly. The self-test takes about a minute to complete and fails only if a hardware failure is detected. If a failure is detected during the self-test, the booting process halts and an error message displays on the system console. *value* can be one of the following: **e** (enabled) Self-test executes. **d** (disabled) Self-test does not execute. The soft front panel must be displayed for the self test to run.
	remote-port-BPS	Selects the default baud rate for the SPU remote port. This is typically set to 1200.
	Scsi-power-up-delay	Controls how many seconds to wait before the boot procedure is started after a power up condition. Some devices must remain undisturbed while executing their power up selftests. *value* can be any hexadecimal number between 0 and FF. To disable this option, specify 0.

Any soft front panel commands not listed here are reserved for CONVEX employees use only.

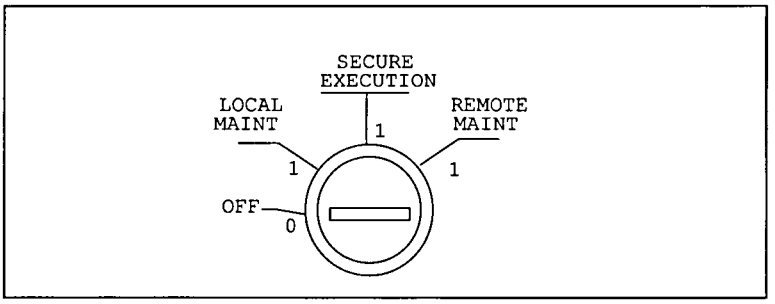
Applying power and initially booting

4

Before you can boot the system, you must apply power to all system cabinets and components. The order in which to do this is described in this chapter.

When the machine is first powered on, it is not running any programs. Once power is applied to system cabinets and components, you can boot the system by turning the keyswitch on the front control panel from the OFF position to one of the ON positions shown in Figure 6.

Figure 6 Front control panel keyswitch



Booting the system initializes the machine, loads a program into main memory, and starts it running. What program starts running when you turn the front control panel keyswitch from the OFF position depends on the new position selected and soft front panel option settings. (This chapter explains these settings.)

Once the system is initially booted, you can boot to other computing environments. How to boot to other computing environments is described in Chapter 5.

Initially booting the system

If system components are powered off, you must apply power to them before you can boot the system. Perform the following steps to apply power to the system cabinets and components and to initially boot the system. Follow each step in sequence. Any attempt at short-cuts can cause serious damage to the system.

Note

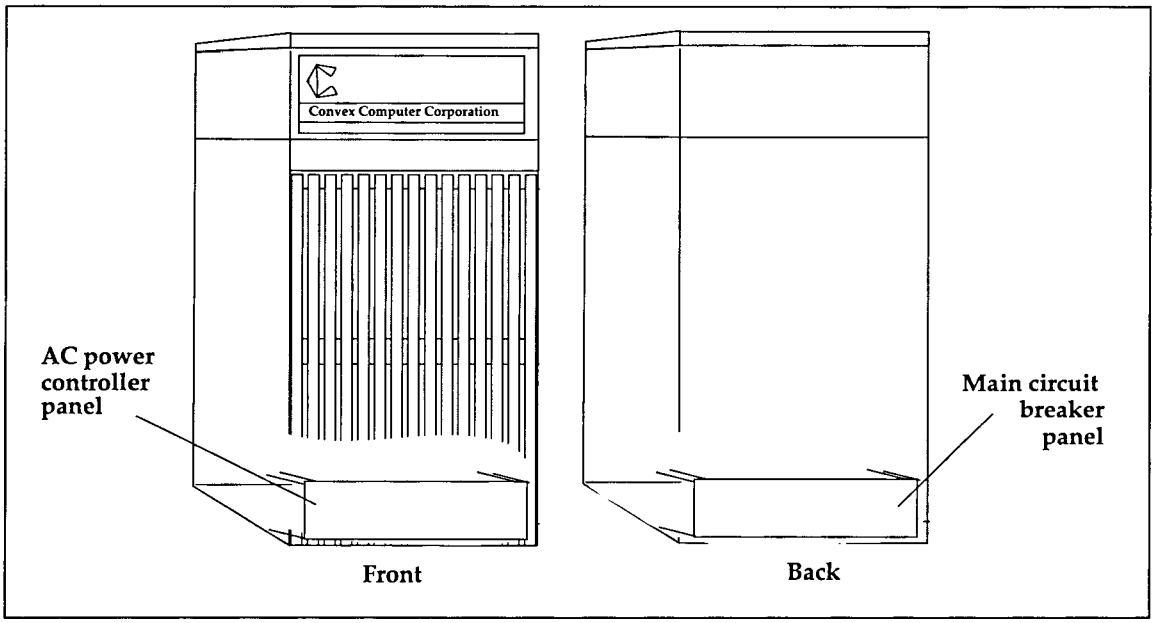
The procedures described in this section assume that all system components, that is, hard disks and tape drives, have been powered down normally. User terminals can be turned on or off at any stage in the procedure.

Step 1: Be sure all breakers and switches are in the OFF position. These are:

- Processor cabinet
  - AC power-controller on processor cabinet
  - Main circuit breaker

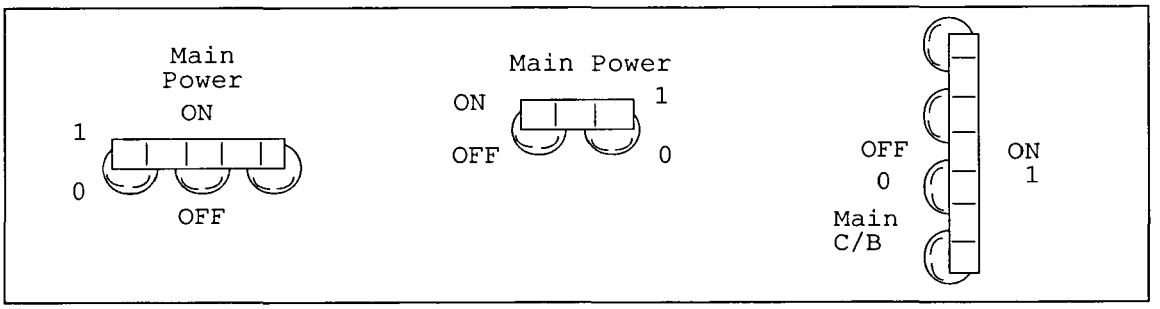
Figure 7 illustrates the location of the breakers and switches in the processor cabinet. Consult the hardware manual for your specific machine for the exact location of these components on your machine.

Figure 7 Location of breakers and switches in processor cabinet



The main power switch on these panels varies from cabinet to cabinet. Figure 8 illustrates some possibilities. Consult the hardware manual for your specific machine for the exact breakers and switches for your machine.

Figure 8 Sample circuit breakers and power switches

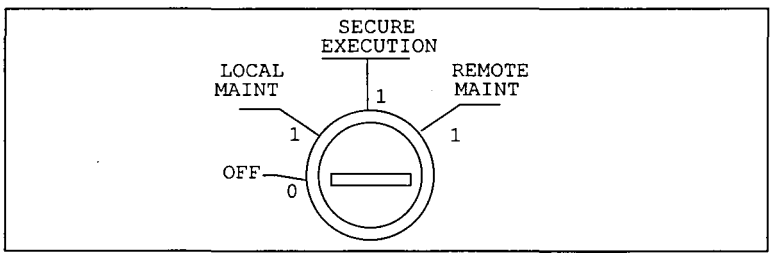


- Expansion cabinets
  - AC power-controller on expansion cabinets
  - Tape drives
  - Disk drives

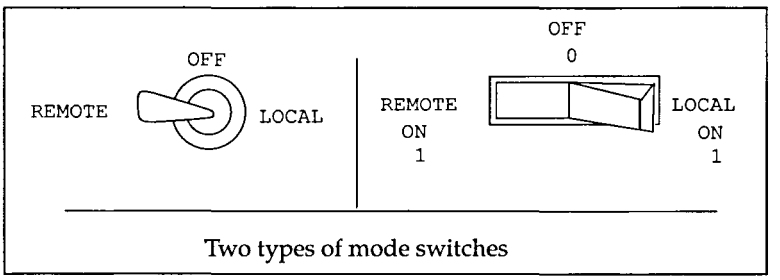
The location of breakers and switches in the expansion cabinets varies from cabinet to cabinet. You will find them in the lower portion of the cabinet, either in the front or back of the cabinet. The main power switch on these panels varies from cabinet to cabinet. Figure 8 illustrates some possibilities. Consult the hardware manual for your specific machine for the exact location of these components for your machine.

Step 2: Be sure there is no tape mounted on the SPU tape drive.

Step 3: Turn the front control panel keyswitch to the OFF position.

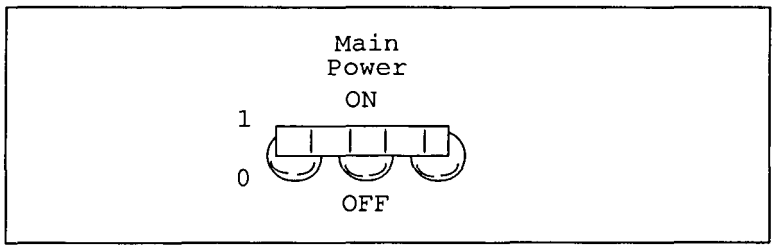


Step 4: Turn the AC power-controller mode switch found on the back of the processor cabinet to the REMOTE position.

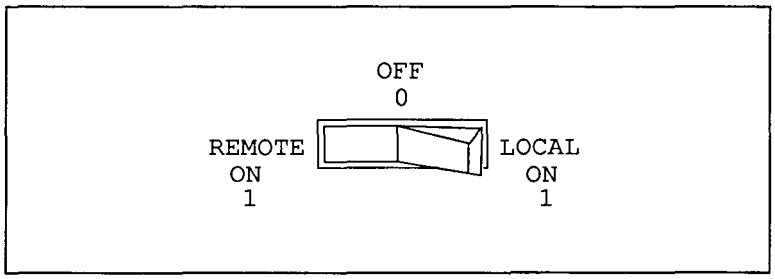


Two types of mode switches

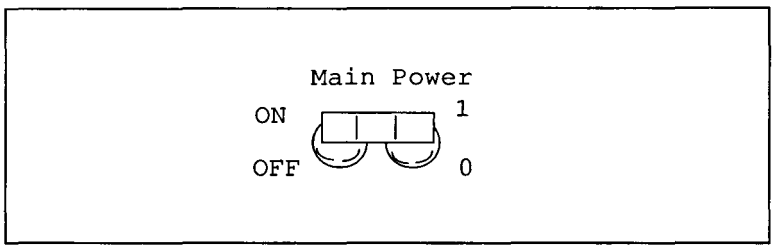
Step 5: Turn the AC power-controller main circuit breaker found on the back of the processor cabinet to the ON position.



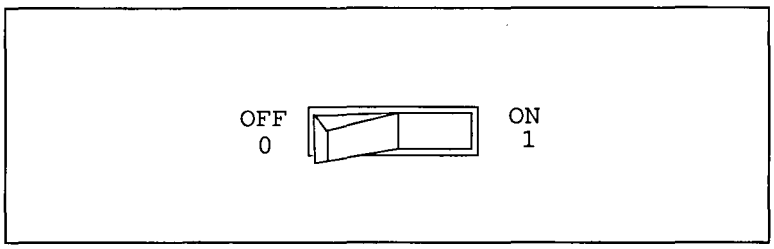
Step 6: If there are additional AC power-controller panels in expansion cabinets, turn the mode switches to the REMOTE position.



Step 7: If there are additional AC power-controller panels in expansion cabinets, turn the circuit breaker switches to the ON position.

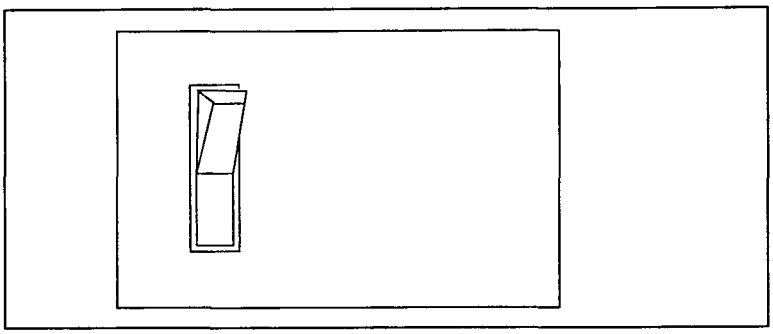


Step 8: Apply power to the tape drives.



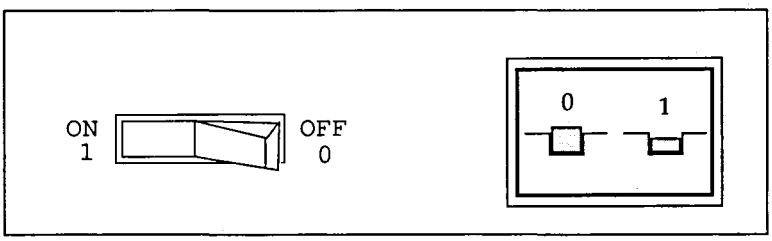
Step 9: Apply power to the disk drive(s). The power switch is on the back of the disk drive. By powering up the disk drives last, you ensure

the disk files are not corrupted by power fluctuations during the power up sequence.



Step 10: If you have trouble powering up the cabinets and components, refer to the "Trouble-shooting the power up" section in this chapter.

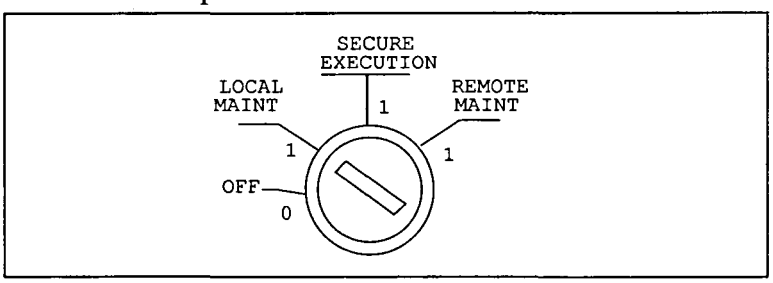
Step 11: Turn the system console power switch to the ON position.



Step 12: The system is now ready to be booted. If you want to boot to the soft front panel, complete Step 13 through Step 14, but skip Step 15. You use the soft front panel to set soft front panel options. See Chapter 3 for details on setting soft front panel options.

If you want to boot to multiuser mode, go to Step 15. Multiuser mode is used for general timesharing.

Step 13: If you do not want to boot to the soft front panel, skip this step and Step 14. To boot to the soft front panel, turn the keyswitch to the LOCAL MAINT position.



Step 14: The system boots to the soft front panel and the soft front panel prompt `(fp)>` appears on the system console. If it does not, refer to the "Trouble-shooting the soft front panel boot" section in this chapter.

Step 15: To boot to multiuser mode, the `mode-of-operation` must be set to `normal-os`. If the mode is not already set to `normal-os`, set it by entering the following command from the `(fp)>` prompt on the system console:

```
set m=n
```

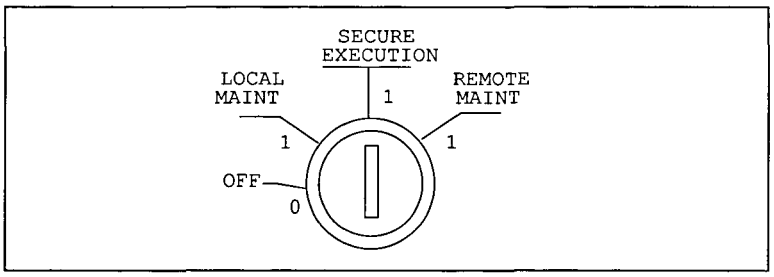
Step 16: To boot to multiuser mode, the `power-up-reboot` soft front panel option must be enabled. If this option is not already enabled, enable it by entering the following command from the `(fp)>` prompt on the system console:

```
set p=e
```

Step 17: To boot to multiuser mode, the `automatic-reboot` soft front panel option must be enabled. If this option is not already enabled, enable it by entering the following command from the `(fp)>` prompt on the system console:

```
set a=e
```

Step 18: Boot to multiuser mode by turning the keyswitch to the SECURE EXECUTION position.



When the keyswitch is turned to the SECURE EXECUTION position, the system performs SPU OS and ConvexOS file checks (`fsck`), initializes memory, and displays status information on the system console for these processes. Once this has successfully completed, the system automatically boots to multiuser mode and displays the multiuser login prompt.

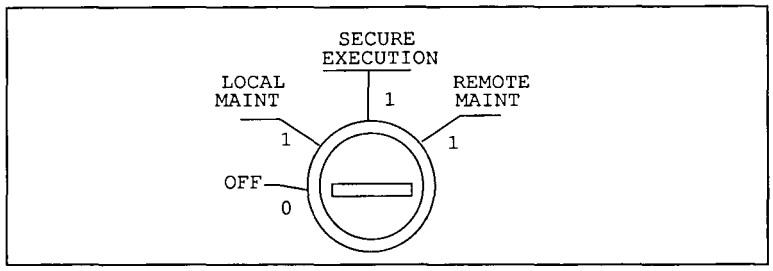
```
login:
```

The system is now ready for use by multiple users.

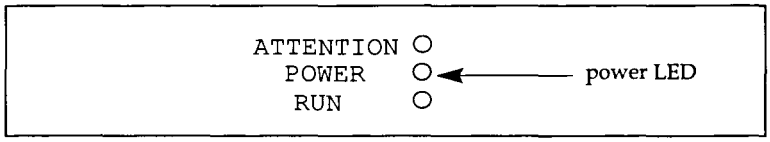
Trouble-shooting the power up

If the power up procedure does not successfully complete, perform the following steps.

Step 1: Check to be sure the keyswitch is not in the OFF position.



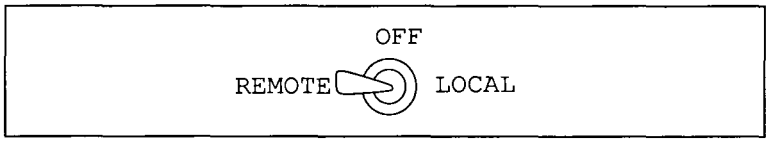
Step 2 Check to be sure the POWER LED on the front control panel is lit.



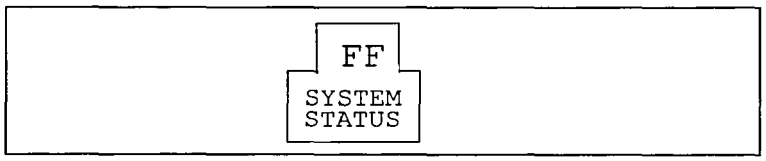
Step 3: Be sure the power lights on all peripherals are lit.

Step 4: Be sure the disk drives spin up.

Step 5: Be sure the mode switch on the AC power-controller panel is in the REMOTE position.



Step 6: Be sure the system status code FF appears on the SYSTEM STATUS display on all models except the C1. Any other status code indicates that the system has a problem.



If all these conditions exist, and the system still fails to power up, turn the keyswitch to the OFF position and repeat the power up procedure. If the system still does not power up properly, contact the CONVEX Technical Assistance Center (TAC).

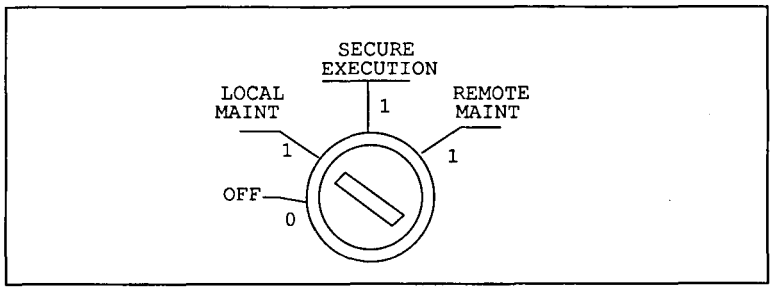
Trouble-shooting the soft front panel boot

The soft front panel program executes whenever you boot the system with the keyswitch turned to the LOCAL MAINT position. Perform the following steps if the soft front panel does not execute when the CPU is first powered up.

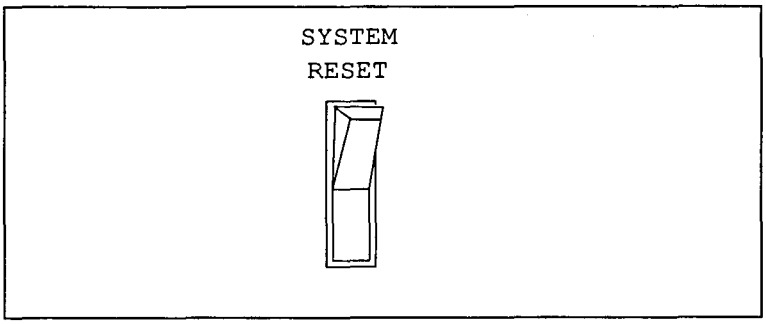
Step 1: Check to be sure the system console is properly connected.

Step 2: Check to be sure the console printer is on and is not out of paper. If the console printer is on, in autoprint mode, and out of paper, the system will eventually crash because SPU buffers will fill up if messages cannot be written to the printer.

Step 3: Set the keyswitch on the front control panel to the LOCAL MAINT position.



Step 4: Press the SYSTEM RESET switch on the front control panel.



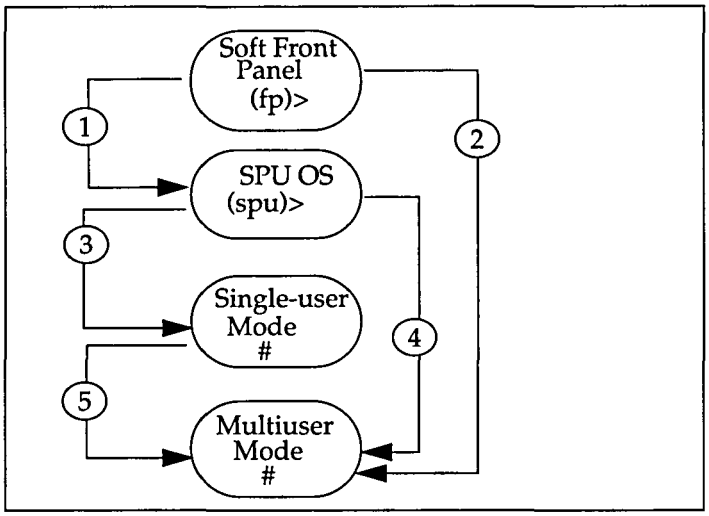
This invokes the soft front panel program and displays the `(fp)>` prompt on the system console.

Booting

5

There is a natural progression from one computing environment to another, starting with the soft front panel and ending with multiuser mode. This progression is illustrated in Figure 9.

Figure 9 Progressing from one environment to another



From the soft front panel, you can boot to:

- SPU OS (path 1)
- Multiuser mode (path 2)

From SPU OS, you can boot to:

- Single-user mode (path 3)
- Multiuser mode (path 4)

From single-user mode, you can boot to:

- Multiuser mode (path 5)

Each of these options are described in the following sections of this chapter.

Booting from the soft front panel

From the soft front panel, you can boot to either multiuser mode or to SPU OS, depending on the value set for the `mode-of-operation` soft front panel option.

SPU OS is used to drive the system console, log system errors, perform system diagnostics, and boot ConvexOS. Multiuser mode is used for general timesharing.

Booting to SPU OS

Use the following procedure to boot to SPU OS. The examples shown are from a C3400 and will vary slightly for other machines.

Step 1: Where the system boots from the soft front panel depends on the value set for the `mode-of-operation` soft front panel option. To boot to SPU OS, the `mode-of-operation` must be set to alternate. If the mode is not already set to alternate, set it by entering the following command from the `(fp)>` prompt on the system console:

```
set m=a
```

Step 2: Boot the system by entering the following command from the `(fp)>` prompt:

```
b
```

Messages displaying the SPU OS version and amount of available memory should appear on the console:

```
SPU OS version 6.0.0.1
Available memory = 3006464 (2936 Kbytes)
```

While booting, the SPU runs integrity checks on the SPU OS file systems. This can take as much as 30 minutes to complete. The following type of messages appears:

```
/dev/rdk0b: 131 files 1r170 blocks 808 free
/dev/rdk0d: 354 files 12480 blocks 468 free
.
.
.
```

If errors are detected by the file integrity checks, the system displays the following error message:

```
RUN fsck MANUALLY
```

Step 3: If you receive the `RUN fsck MANUALLY` message described in the previous step, manually run the `fsck` program on the file systems referenced in the error message. See *Managing ConvexOS: Operations Guide* or the SPU OS fsck(8) man page for procedures on running `fsck`.

Caution

The file systems may be irreparably damaged if you do not run `fsck`.

Step 4: If the `fsck` program fails at this point, call the TAC.

Step 5: Once all file systems are checked, file systems are mounted. The system displays the following messages on the system console:

```
SPU filesystem verified
Mounted /mnt on /dev/dk0d
.
.
.
```

SPU OS is booted when you receive the following messages:

```
SPU OS booted Nov 11 13:33 1991 Freq: 60 Hz.
Mon Nov 11 13:37:06 CST 1991
```

The following menu appears.

```
multi      Boot multi-user <default>
single     Boot single-user
mini       Boot Single-user using miniroot
quit       Exit to SPU shell
```

Step 6: Exit to the SPU shell by entering

`quit`

You must enter `quit` exactly as shown as abbreviations are not allowed here. The soft front panel transfers control to a boot program read from the SPU disk.

Booting to multiuser mode

Use the following procedure to boot to multiuser mode. The examples shown are from a C3400 and will vary slightly for other machines.

Step 7: To boot to multiuser mode, the `mode-of-operation` must be set to `normal-os`. If the mode is not already set to `normal-os`, set it by entering the following command from the `(fp)>` prompt on the system console:

```
set m=n
```

Step 8: To boot to multiuser mode, the `power-up-reboot` soft front panel option must be enabled. If this option is not already enabled, enable it by entering the following command from the `(fp)>` prompt on the system console:

```
set p=e
```

Step 9: To boot to multiuser mode, the `automatic-reboot` soft front panel option must be enabled. If this option is not already enabled, enable it by entering the following command from the `(fp)>` prompt on the system console:

```
set a=e
```

Step 10.: Boot the system by entering the following command from the `(fp)>` prompt:

```
b
```

The soft front panel transfers control to a boot program read from the SPU disk. Messages displaying the SPU OS version and amount of available memory should appear on the console:

```
SPU OS version 6.0.0.1
Available memory = 3006464 (2936 Kbytes)
```

Step 11: If the system fails at this point, press the system reset button and repeat Step 10; otherwise, go on to Step 12.

Step 12: While booting, the SPU runs integrity checks on the SPU OS file systems. This can take as much as 30 minutes to complete. The following type of messages appears:

```
/dev/rdk0b: 131 files 1r170 blocks 808 free
/dev/rdk0d: 354 files 12480 blocks 468 free
.
.
.
```

If errors are detected by the file integrity checks, the system displays the following error message:

```
RUN fsck MANUALLY
```

Step 13: If you receive the `RUN fsck MANUALLY` message described in the previous step, manually run the `fsck` program on the file systems referenced in the error message. See *Managing ConvexOS: Operations Guide* or the SPU OS fsck(8) man page for procedures on running `fsck`.

Caution

The file systems may be irreparably damaged if you do not run `fsck`.

Step 14: If the `fsck` program fails at this point, call the TAC.

Step 15: Once all file systems are checked, file systems are mounted. The system displays the following messages on the system console:

```
SPU filesystem verified
Mounted /mnt on /dev/dk0d
.
.
.
```

SPU OS is booted when you receive the following messages:

```
SPU OS booted Nov 11 13:33 1991 Freq: 60 Hz.
Mon Nov 11 13:37:06 CST 1991
```

The system continues by booting ConvexOS. ConvexOS is booted when you receive the following message:

```
[CPU02@13:45:45] All Rights Reserved.
```

The system continues booting to multiuser mode. When the multiuser boot completes, the multiuser login prompt appears:

```
login:
```

The system is now ready for use by multiple users.

Booting from SPU OS

If you are booted to SPU OS, you can elect to boot to single-user mode or multiuser mode. The procedures to boot to either of these environments are provided in the following sections.

Caution

You can be at the SPU prompt by booting from the soft front panel to SPU OS or by pressing CTRL-p from ConvexOS. CTRL-p from ConvexOS puts you at the SPU prompt to run commands from SPU OS, but ConvexOS remains running. Be careful not to boot ConvexOS from the SPU if it is already running. You can check to be sure it is not running by pressing CTRL-d at the SPU prompt. If ConvexOS is running, you receive a ConvexOS prompt. If it is not running, you receive the SPU prompt again. If you receive the SPU prompt, you can be assured ConvexOS is not running and you can perform either of the following procedures to boot ConvexOS.

Booting to single-user mode

Use this procedure to boot to single-user mode from SPU OS. Single-user mode disables all user logins and all terminal processing except the system console. Because users are unable to use the system while in single-user mode, it is considered safer and more effective than using multiuser mode to perform administrative tasks such as:

- Software installation
- File system checks
- Recovery from crashes
- Mounting file systems
- Running system checks
- Disk striping

Caution

Do not write-protect disks and then boot or run ConvexOS. If this should occur, do not write-enable the disks until after the system has been completely shut down and the sysreset program has been run. Failure to observe this CAUTION can result in corrupted files.

Step 1: This procedure begins from the SPU. Enter the following command at the `(spu)>` prompt:

```
boot single
```

ConvexOS boots to single-user mode and displays the following message:

```
erase ^H, kill ^U, intr ^C
 #
```

The first line shows the console settings for the character erase (`^H`), line erase (`^U`), and interrupt characters (`^C`), respectively. The pound sign (#) is the root prompt from the command interpreter, `sh`. It differs from the standard `sh` prompt, $, that displays for users who are not the superuser.

Step 2: Perform consistency checks on the disks using the `preen` command. Enter

```
preen
```

The `preen` command invokes the `fsck` program for each local file system listed in the /etc/fstab file. The /etc/fstab file contains descriptions of file system layouts.

After checking each file system, `fsck` prints a line that lists the device name and the number of files found in that file system, and displays a message for each inconsistency it discovers.

Step 3: If errors occur on the root file system that `preen` will fix, the system automatically reboots once they are fixed. If errors are found on the root file system that `preen` will not fix, the following message appears:

```
RUN fsck MANUALLY
```

If this happens, repair the root file system by running `fsck` manually. You should run `fsck` until it stops failing. See the fsck(8) man page for the details on running `fsck`.

Step 4: Once `fsck` completes, reboot using the following command:

```
reboot -n
```

The `reboot` command reboots the system to multiuser mode. There is no need to reboot if errors are found in file systems other than the root file system.

See the preen(8) man page for additional information on the `preen` command.

Booting to multiuser mode

Use the following procedure to boot to multiuser mode from SPU OS. Multiuser mode is used for general timesharing. Invoking the multiuser system starts the daemons and runs `init` to enable logins on all terminals and to mount all file systems.

Step 1: This procedure begins from the SPU prompt, `(spu)>`. Enter the following command on the system console:

```
boot
```

The system automatically boots to multiuser mode.

When ConvexOS boots, the system attempts to run the `preen` utility. The `preen` command invokes the `fsck` program for each local file system listed in the /etc/fstab file.

After checking each file system, `fsck` prints a line that lists the device name and the number of files found in that file system and displays a message for each inconsistency it discovers.

Step 2: If errors that `preen` will fix occur on the root file system, the system automatically reboots once they are fixed. If errors that `preen` will not fix occur on the root file system, the system remains in single-user mode. The automatic reboot fails because `preen` failed. Run `fsck` manually on the corrupted file system until it stops failing. Then reboot the system by entering:

```
/etc/reboot
```

Step 3: If errors that `preen` will fix occur on a file system other than the root file system, the boot continues after the errors are fixed. If errors that `preen` will not fix occur on any file system other than root, the system remains in single-user mode. The system will not go to multiuser mode from single-user mode if file systems are corrupted. Run `fsck` manually on the corrupted file systems. You should run `fsck` until it stops failing. Then press **CTRL-d** to move to multiuser mode.

Step 4: If all file systems are successfully checked without error, the system displays the multiuser login prompt:

```
login:
```

The system is now ready for use by multiple users.

Booting from single-user mode to multiuser mode

From single-user mode, you can boot to multiuser mode. Use this procedure to boot to multiuser mode from single-user mode.

Step 1: Change your working directory to the root file system. Enter

```
cd /
```

Step 2: This procedure begins from the single-user mode. Unmount all file systems by entering the following command on the system console:

```
umount -a
```

Step 3: If the system crashed or hung, or if you had problems going to single-user mode from multiuser mode, run `preen` to check the file systems. Enter

```
preen
```

Step 4: Press **CTRL-d** at the single-user prompt.

The system automatically mounts all file systems specified in the /etc/fstab file, boots to the multiuser mode, and displays the multiuser login prompt:

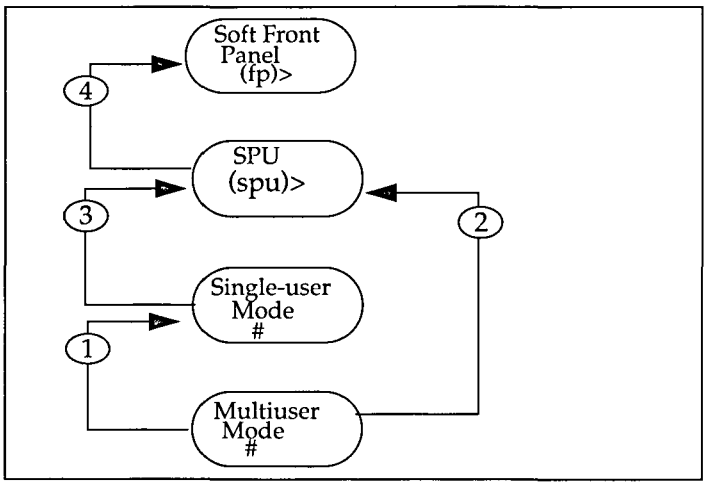
```
login:
```

The system is now ready for use by multiple users.

Shutting down

After the system is booted to multiuser mode, it is sometimes necessary to shut the system down to another computing environment. As in the boot process, there is a natural digression from one computing environment to another, starting with multiuser mode and ending with the soft front panel. This digression is illustrated in Figure 10.

Figure 10 Digressing from one environment to another



From multiuser mode, you can shut down to:

- Single-user mode (path 1)
- SPU OS (path 2)

From single-user mode, you can shut down to:

- SPU OS (path 3)

From SPU OS, you can shut down to:

- Soft front panel (path 4)

Each of these options are described in the following sections of this chapter.

Shutting down from multiuser mode

If you are in multiuser mode and want to mount a new file system or perform backups, you must first shut down to single-user mode. If the system hangs, normal recovery procedures require you to shut down the system gracefully to SPU OS before trying to reboot the system. The following sections describe how to shut down the system from multiuser mode.

Shutting down to single-user mode

Use this procedure to shut down from multiuser mode to single-user mode.

Step 1: This procedure begins from multiuser mode. Use the `shutdown` command to shut down to single-user mode. Include a message to users informing them of the impending shut down and define the amount of time to wait before shut down occurs. For example, enter

```
shutdown +10 "time for dumps"
```

This command periodically sends the specified message to all users logged into the system and then shuts down to single-user mode after 10 minutes. All user processes are terminated and a single-user shell prompt is displayed. File systems remain mounted. (See the shutdown(8) man page for more details on this command.)

Step 2: Wait until the following message appears:

```
erase ^H, KILL ^U, intr ^C
```

Step 3: Once this message displays, ensure that the information on the disks is up-to-date by entering

```
sync;sync
```

Shutting down to SPU OS

Use this procedure to shut down from multiuser mode to SPU OS.

Step 1: This procedure begins from multiuser mode. Use the `shutdown -h` command to shut down to SPU OS. Include a message to users informing them of the impending shut down and define the amount of time to wait before shut down occurs. For example, enter

```
shutdown -h +10 "time to dump"
```

This command periodically sends the specified message to all users logged into the system and then shuts down to SPU OS after 10 minutes. All user processes are terminated, ConvexOS is halted, and the following messages are displayed on the system console:

```
[CPU hh:mm:ss] syncing disks...
[CPU hh:mm:ss] done
[CPU hh:mm:ss] halting in tight loop;type
 ''sysreset'' to halt
```

Step 2: When the `(spu)>` prompt appears, enter

```
sysreset
```

Shutting down from single-user to SPU OS

Use this procedure to shut down from single-user mode to SPU OS. The `shutdown` command terminates active processes and halts ConvexOS.

Step 1: From single-user mode, use the `halt` command to shut down to SPU OS. To do this, enter:

```
halt
```

The system displays the following messages on the system console:

```
(spu)> [CPU hh:mm:ss] syncing disks...
(spu)> [CPU hh:mm:ss] done
(spu)> [CPU hh:mm:ss] halting in tight loop;
 type "sysreset" to halt
```

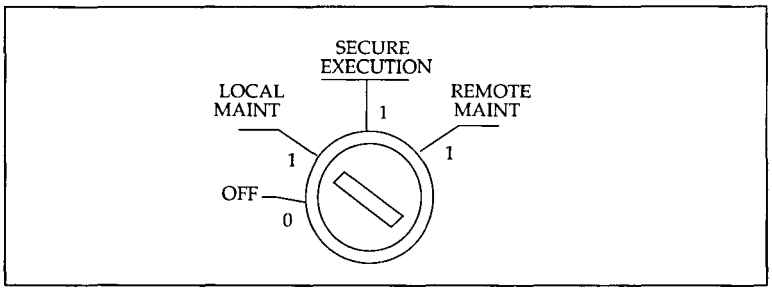
Step 2: When the `(spu)>` prompt appears, enter

```
sysreset
```

Shutting down from SPU OS to soft front panel

Normal system recovery procedures require the system manager to shut down the system gracefully to SPU OS before booting to multiuser or single-user mode. Once at SPU OS, if the system cannot be booted, you must shut down to the soft front panel to reboot the system. Use the following procedure to do this.

Step 1: This procedure begins from the SPU OS prompt, `(spu)>`. Turn the keyswitch to the LOCAL MAINT position.



Step 2: Be sure there is no tape cartridge on the SPU tape drive.

Step 3: Enter the following command at the `(spu)>` prompt:

```
/etc/reboot
```

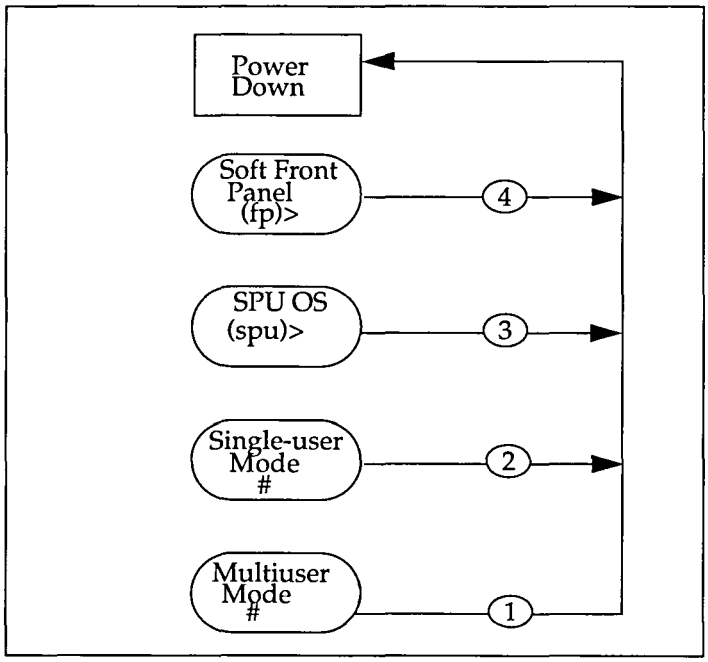
The system displays the soft front panel prompt, `(fp)>`.

Powering down

7

It is sometimes necessary to power the system down to service the computer. The procedure you use to power the system down depends on the computing environment you are in. Figure 11 illustrates the computing environments from which you can power down.

Figure 11 Powering the system down



You can power down from:

- Multiuser mode (path 1)
- Single user mode (path 2)
- SPU OS (path 3)
- Soft front panel (path 4)

The following sections of this chapter describe how to power the system down from each computing environment.

Power down from multiuser mode

Use this procedure to power down from multiuser mode.

Step 1: If you do not have CXbatch running on your machine, skip this step and go to Step 5. If you have CXbatch running on your machine, you must shut down CXbatch before shutting down the system in order to checkpoint any checkpointable jobs. To do this, you must first start the qmgr utility. Start the qmgr utility by entering

```
qmgr
```

The qmgr prompt appears:

```
Mgr:
```

Step 2: Shut down CXbatch and checkpoint checkpointable jobs by entering

```
shutdown
```

Step 3: Exit the qmgr uitlity by entering

```
exit
```

Step 4: Shut down SPU OS using the `shutdown -h` command. Include a message to users informing them of the impending shut down and define the amount of time to wait before shut down occurs. For example, enter

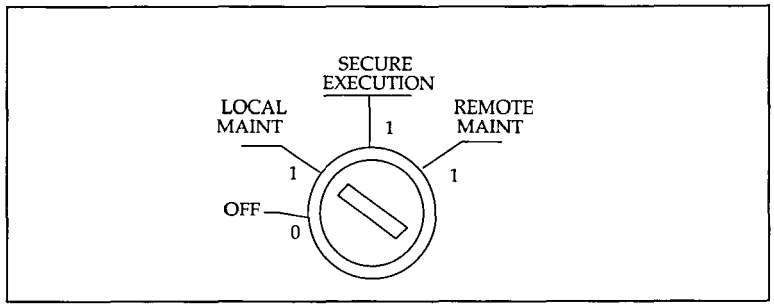
```
shutdown -h +10 "time to take dumps"
```

This command periodically sends the specified message to all users logged into the system and then shuts down to SPU OS after 10 minutes. All user processes are terminated, ConvexOS is halted, and the following messages are displayed on the system console:

```
[CPU hh:mm:ss] syncing disks...
[CPU hh:mm:ss] done
[CPU hh:mm:ss] halting in tight loop;type
 ''sysreset'' to halt
```

The system then displays the SPU prompt, `(spu)>`.

Step 5: Turn the keyswitch to the LOCAL MAINT position.



Step 6: Execute the `pwrdwn` command. Enter

```
pwrdwn
```

SPU OS terminates and the soft front panel executes. The system displays the following message on the system console:

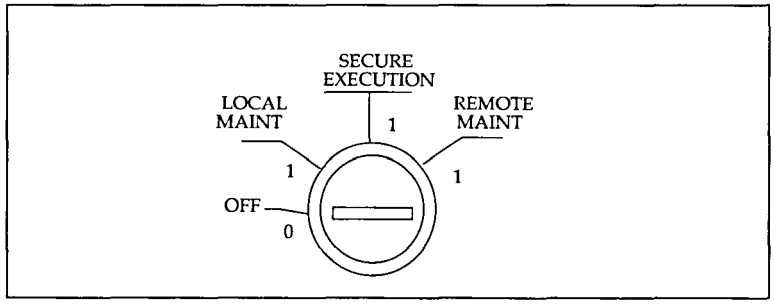
```
pwrdwn: Ready for power down. ^D to abort.
```

Step 7: If you want to abort the power down, press **CTRL-d**. Power down aborts and the system displays the following message:

```
pwrdwn: Power down aborted.
```

Then the `(spu)>` prompt appears.

Step 8: If you want to continue powering down, turn the front panel keyswitch to the OFF position.



Power down from single-user mode

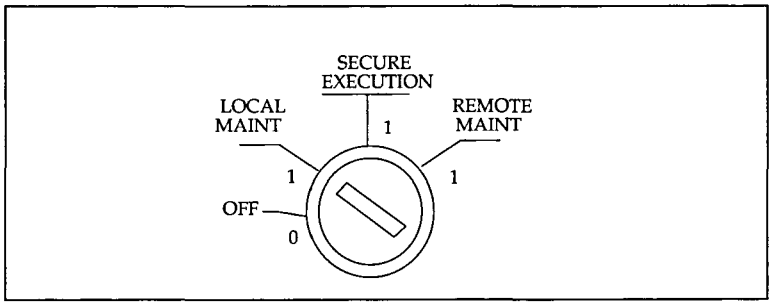
Use this procedure to power down from single-user mode.

Step 1: Use the `shutdown -h` command to gracefully shut down from single-user mode to SPU OS. For example, enter

```
shutdown -h +10 "time to dump"
```

The `shutdown` command terminates active processes and halts ConvexOS. The system then displays the SPU prompt, `(spu)>`.

Step 2: Turn the keyswitch to the LOCAL MAINT position.



Step 3: Execute the `pwrdwn` command. Enter

```
pwrdwn
```

The `pwrdwn` command terminates SPU OS and the system displays the following message on the system console:

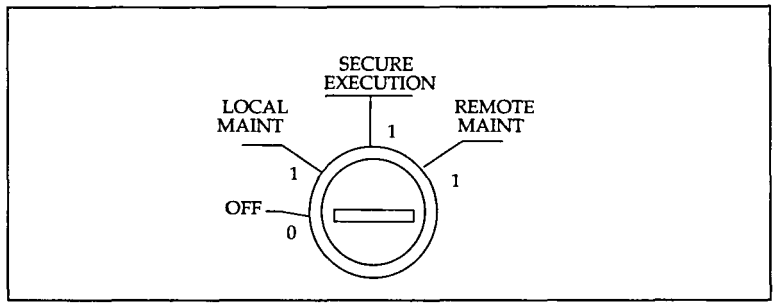
```
pwrdwn: Ready for power down. ^D to abort.
```

Step 4: If you want to abort the power down, press **CTRL-d**. Power down aborts and the system displays the following message:

```
pwrdwn: Power down aborted.
```

Then the `(spu)>` prompt appears.

Step 5: If you want to continue powering down, turn the front panel keyswitch to the OFF position.

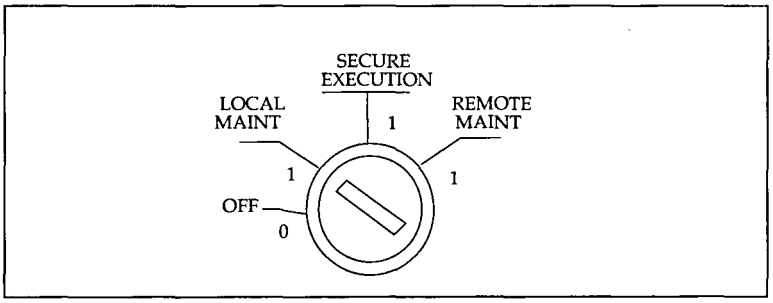


Power down from SPU OS

Use this procedure to power down from SPU OS. ConvexOS must be halted before performing this procedure.

Step 1: Ensure that ConvexOS is halted. You can tell that ConvexOS is halted if the `(spu)>` prompt is displayed on the console. If it is not, press **CTRL-p**.

Step 2: Turn the keyswitch to the LOCAL MAINT position..



Step 3: Execute the `pwrdwn` command. Enter

```
pwrdwn
```

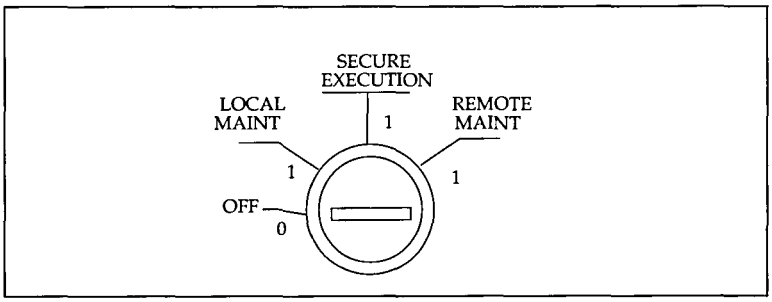
The `pwrdwn` command terminates SPU OS and the system displays the following message on the system console:

```
pwrdwn: Ready for power down. ^D to abort.
```

Step 4: If you want to abort the power down, press **CTRL-d**. Power down aborts, the system displays the following message, and returns to the `(spu)>` prompt:

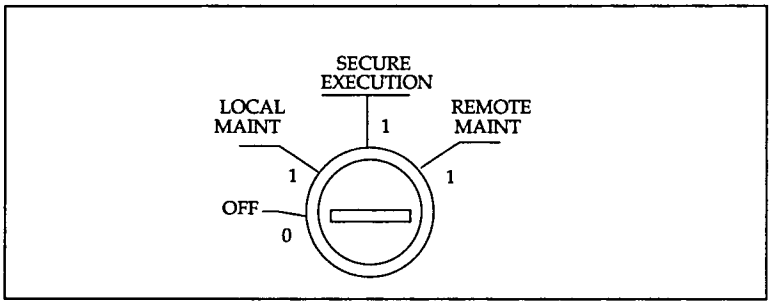
```
pwrdwn: Power down aborted.
```

Step 5: If you want to continue powering down, turn the front panel keyswitch to the OFF position.



Power down from soft front panel

To power down from the soft front panel, turn the front control panel keyswitch to the OFF position.



Backing up SPU files

8

Back up the files on the SPU disk when:

- A new software release is installed
- You modify a file

Perform the following steps to back up file systems on the SPU disk.

Step 1: Place a cartridge tape in the SPU tape drive.

Step 2: If SPU OS is booted, you will see the `(SPU)>` prompt. If ConvexOS is booted you must get the `(SPU)>` prompt by pressing **CTRL-p** on the system console.

Step 3: Determine whether or not you have a QIC tape drive. If you do not know, you can find out by entering

```
ll /dev/rmt0
```

If a message is returned that says it is not found, you do not have a QIC drive.

If you do not have a QIC tape drive, proceed with Step 4 but skip Step 5.

If you have a QIC tape drive, skip Step 4 and go to Step 5.

Step 4: If you do not have a QIC tape drive, format the tape by entering

```
ctutil fmt
```

Now skip Step 5 and go on to Step 6.

Step 5: If you have a QIC tape drive, rewind the tape by entering

```
mt rewind
```

Step 6: Dump the file systems on the SPU to the tape by entering:

```
/etc/backup
```

This backs up all file systems on the SPU and creates a bootable backup tape. You can boot from this backup tape if necessary.

Step 7: Once the dump completes, remove the tape from the tape drive.

Step 8: Label and archive the tape.

SPU man pages

A

This appendix contains the man pages that describe the features of the SPU. The man pages are organized into the following sections:

- Commands
- File formats
- System management

Commands

This section describes publicly accessible commands in alphabetic order.

NAME

intro – introduction to commands

DESCRIPTION

This section describes publicly accessible commands in alphabetic order. The word 'local' at the foot of a page means that the command is not intended for general distribution.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of 'normal' termination) one supplied by the program, see **wait(2)** and **exit(2)**. The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously 'exit code', 'exit status' or 'return code', and is described only where special conventions are involved.

NAME

adb – debugger

SYNOPSIS

adb [**–w**] [objfil [corfil]]

DESCRIPTION

Adb is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of UNIX programs.

Objfil is normally an executable program file, preferably containing a symbol table; if not then the symbolic features of **adb** cannot be used although the file can still be examined. The default for *objfil* is **b.out**. *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is *core*.

Requests to **adb** are read from the standard input and responses are to the standard output. If the **–w** flag is present then both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using **adb**. **Adb** ignores QUIT; INTERRUPT causes return to the next **adb** command.

In general requests to **adb** are of the form

[*address*] [, *count*] [*command*] [;]

If *address* is present then *dot* is set to *address*. Initially *dot* is set to 0. For most commands *count* specifies how many times the command will be executed. The default *count* is 1. *Address* and *count* are expressions.

The interpretation of an address depends on the context it is used in. If a subprocess is being debugged then addresses are interpreted in the usual way in the address space of the subprocess. For further details of address mapping see ADDRESSES.

EXPRESSIONS

. The value of *dot*.

\+ The value of *dot* incremented by the current increment.

^ The value of *dot* decremented by the current increment.

" The last *address* typed.

integer An octal number if *integer* begins with a 0; otherwise, a hexadecimal number.

'cccc ' The ASCII value of up to 4 characters. \ may be used to escape a '.

< name

The value of *name*, which is either a variable name or a register name. **Adb** maintains a number of variables (see VARIABLES) named by single letters or digits. If *name* is a register name then the value of the register is obtained from the system header in *corfil*. The register names are **d0 ... d7 a0 ... a6 sp pc ps**.

symbol A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The value of the *symbol* is taken from the symbol table in *objfil*. An initial _ or ~ will be prepended to *symbol* if needed.

_ symbol

In C, the 'true name' of an external symbol begins with _. It may be necessary to utter this name to disinguish it from internal or hidden variables of a program.

routine.name

The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*.

(*exp*) The value of the expression *exp*.

Monadic operators

**exp* The contents of the location addressed by *exp* in *corfil*.

@ exp The contents of the location addressed by *exp* in *objfil*.

-exp Integer negation.

~exp Bitwise complement.

Dyadic operators are left associative and are less binding than monadic operators.

e1+e2 Integer addition.

e1-e2 Integer subtraction.

*e1*e2* Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

e1#e2 *E1* rounded up to the next multiple of *e2*.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands '?' and '/' may be followed by '*'; see ADDRESSES for further details.)

?f Locations starting at *address* in *objfil* are printed according to the format *f*.

/f Locations starting at *address* in *corfil* are printed according to the format *f*.

=f The value of *address* itself is printed in the styles indicated by the format *f*. (For **i** format '?' is printed for the parts of the instruction that reference subsequent words.)

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format *dot* is incremented temporarily by the amount given for each format letter. If no format is given then the last format is used. The format letters available are as follows.

o 2 Print 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O 4 Print 4 bytes in octal.
q 2 Print in signed octal.
Q 4 Print long signed octal.
d 2 Print in decimal.
D 4 Print long decimal.
x 2 Print 2 bytes in hexadecimal.
X 4 Print 4 bytes in hexadecimal.
u 2 Print as an unsigned decimal number.
U 4 Print long unsigned decimal.
b 1 Print the addressed byte in octal.
c 1 Print the addressed character.
C 1 Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s *n* Print the addressed characters until a zero character is reached.
S *n* Print a string using the @ escape convention. *n* is the length of the string including its zero terminator.
Y 4 Print 4 bytes in date format (see **ctime(3)**).
i n Print as 68000 instructions. *n* is the number of bytes occupied by the

instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.

a 0 Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

p 2 Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.

t 0 When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.

r 0 Print a space.

n 0 Print a newline.

"..." 0 Print the enclosed string.

^ *Dot* is decremented by the current increment. Nothing is printed.

\+ *Dot* is incremented by 1. Nothing is printed.

\- *Dot* is decremented by 1. Nothing is printed.

newline

If the previous command temporarily incremented *dot*, make the increment permanent. Repeat the previous command with a *count* of 1.

[?/]**l** *value mask*

Words starting at *dot* are masked with *mask* and compared with *value* until a match is found. If **L** is used then the match is for 4 bytes at a time instead of 2. If no match is found then *dot* is unchanged; otherwise *dot* is set to the matched location. If *mask* is omitted then –1 is used.

[?/]**w** *value ...*

Write the 2-byte *value* into the addressed location. If the command is **W**, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]**m** *b1 e1 f1*[?/]

New values for (*b1, e1, f1*) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second segment (*b2, e2, f2*) of the mapping is changed. If the list is terminated by '?' or '/' then the file (*objfil* or *corfil* respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to *objfil*.)

>*name* *Dot* is assigned to the variable or register named.

! A shell is called to read the rest of the line following '!'.

$*modifier*

Miscellaneous commands. The available *modifiers* are:

<*f* Read commands from the file *f* and return.

>*f* Send output to the file *f*, which is created if it does not exist.

r Print the general registers and the instruction addressed by **pc**. *Dot* is set to **pc**.

b Print all breakpoints and their associated counts and commands.

c C stack backtrace. If *address* is given then it is taken as the address of the current frame (instead of **a6**). If **C** is used then the names and (16 bit) values of all automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.

e The names and values of external variables are printed.

w Set the page width for output to *address* (default 80).

s Set the limit for symbol matches to *address* (default 255).

d All integers input are regarded as decimal.
q Exit from *adb*.
v Print all non-zero variables in hexadecimal.
m Print the address map.
x All integers input are regarded as hexidecimal.

:*modifier*

Manage a subprocess. Available modifiers are:

bc Set breakpoint at *address*. The breakpoint is executed *count*–1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command sets *dot* to zero then the breakpoint causes a stop.

d Delete breakpoint at *address*.

r Run *objfil* as a subprocess. If *address* is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

c*s* The subprocess is continued with signal *s* c *s*, see **signal(2)**. If *address* is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

s*s* As for **c** except that the subprocess is single stepped *count* times. If there is no current subprocess then *objfil* is run as a subprocess as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

k The current subprocess, if any, is terminated.

VARIABLES

Adb provides a number of variables. Named variables are set initially by *adb* but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

On entry the following are set from the system header in the *corfil*. If *corfil* does not appear to be a *core* file then these values are set from *objfil*.

b The base address of the data segment.
d The data segment size.
e The entry point.
m The 'magic' number (0405, 0407, 0410 or 0411).
s The stack segment size.
t The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*) and the *file address* corresponding to a written *address* is calculated as follows.

$b1 \leq address < e1 \Rightarrow file\ address = address + f1 - b1$, otherwise,

$$b2 \leq address < e2 \Rightarrow file\ address = address + f2 - b2,$$

otherwise, the requested *address* is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a ? or / is followed by an * then only the second triple is used.

The initial setting of both mappings is suitable for normal *b.out* and *core* files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size and *f1* is set to 0; in this way the whole file can be examined with no address translation.

So that *adb* may be used on large files all appropriate values are kept as signed 32 bit integers.

FILES

/dev/mem
/dev/swap
b.out
core

SEE ALSO

ptrace(2)
b.out(5)
core(5)

DIAGNOSTICS

Adb appears when there is no current command or format. Comments are made about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned non-zero status.

BUGS

A breakpoint set at the entry point is not effective on initial entry to the program.

When single stepping, system calls do not count as an executed instruction.

Local variables whose names are the same as an external variable may foul up the accessing of the external.

NAME

cat – catenate and print

SYNOPSIS

cat file ...

DESCRIPTION

Cat reads each *file* in sequence and writes it on the standard output. Thus

cat file

prints the file and

cat file1 file2 >file3

concatenates the first two files and places the result on the third.

If no *file* is given, or if the argument '–' is encountered, **cat** reads from the standard input. Output is buffered in 512-byte blocks unless the standard output is a terminal.

SEE ALSO

cp(1)

BUGS

Beware of 'cat a b >a' and 'cat a b >b', which destroy input files before reading them.

NAME

cd – change working directory

SYNOPSIS

cd directory

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *cd* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

SEE ALSO

sh(1)
pwd(1)
chdir(2)

NAME

chmod – change mode

SYNOPSIS

chmod mode file ...

DESCRIPTION

The mode of each named file is changed according to *mode*, which may be absolute or symbolic. An absolute *mode* is an octal number constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit, see **chmod(2)**
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic *mode* has the form:

[*who*] *op permission* [*op permission*] ...

The *who* part is a combination of the letters **u** (for user's permissions), **g** (group) and **o** (other). The letter **a** stands for **ugo.** If *who* is omitted, the default is *a* but the setting of the file creation mask (see **umask(2)**) is taken into account.

Op can be + to add *permission* to the file's mode, – to take away *permission* and = to assign *permission* absolutely (all other bits will be reset).

Permission is any combination of the letters **r** (read), **w** (write), **x** (execute), **s** (set owner or group id) and **t** (save text – sticky). Letters **u, g** or **o** indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with = to take away all permissions.

The first example denies write permission to others, the second makes a file executable:

```
chmod o-w file
chmod +x file
```

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter **s** is only useful with **u** or **g.**

Only the owner of a file (or the super-user) may change its mode.

SEE ALSO

ls(1)
chmod(2)
stat(2)
umask(2)

NAME

clear – clear terminal screen

SYNOPSIS

clear

DESCRIPTION

Clear clears your screen if this is possible. It looks in the environment for the terminal type and then in */etc/termcap* to figure out how to clear the screen.

FILES

/etc/termcap terminal capability data base

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] file1 file2

DESCRIPTION

The two files are compared. (If *file1* is '–', the standard input is used.) Under default options, **cmp** makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

Options:

–l Print the byte number (decimal) and the differing bytes (octal) for each difference.

–s Print nothing for differing files; return codes only.

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

cp – copy

SYNOPSIS

cp file1 file2

cp file ... directory

DESCRIPTION

File1 is copied onto *file2*. The mode and owner of *file2* are preserved if it already existed; the mode of the source file is used otherwise.

In the second form, one or more *files* are copied into the *directory* with their original file-names.

Cp refuses to copy a file onto itself.

SEE ALSO

cat(1)
mv(1)

NAME

csh – a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [**–cefinstvVxX**] [arg ...]

DESCRIPTION

csh is a first implementation of a command language interpreter incorporating a history mechanism (see **History Substitutions**), job control facilities (see **Jobs**), and a C-like syntax.

An instance of *csh* begins by executing commands from the file *.cshrc* in the *home* directory of the invoker. Typical items to include in a *.cshrc* file are aliases and other variable settings. There are certain advantages if you avoid setting a lot of aliases or your prompt when the current shell is not interactive. This can be accomplished by the inclusion of the line

if (! $?prompt) exit

in your *.cshrc* file immediately after the lines that you want to have executed *every* time a shell is invoked. If this is a login shell, then *csh* also executes commands from the global login file */etc/login*, and from the home directory file *.login*, in that order. **Note: *csh* as the login shell under SPU OS is not currently supported.** It is typical for users on crt's to put the command **stty crt** in their *.login* file and to invoke *tset*(1) there.

In the normal case, the shell then begins reading commands from the terminal, prompting with % . Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into *words;* this sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates, it executes commands first from the file *.logout* in the users home directory and then from the global logout file */etc/logout*.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters &, |, ;, <, >, (, and) form separate words. If doubled in &&, ||, <<, or >>, these pairs form single words. These parser metacharacters may be made part of other words. To prevent their special meaning, precede them with \. A newline preceded by a \ is equivalent to a blank.

In addition, strings enclosed in matched pairs of quotations ', `, or ", form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of ' or " characters, a newline preceded by a \ gives a true newline character.

When the shell's input is not a terminal, the character # introduces a comment that continues to the end of the input line. This special meaning is prevented when the # is preceded by \ and is in quotations using `, ', and ".

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by | characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ;, and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an &.

Any of the above may be placed in (' ') to form a simple command that may be a component of a pipeline, etc. It is also possible to separate pipelines with || or && indicating, as in the C

language, that the second is to be executed only if the first fails or succeeds respectively. (See **Expressions**.)

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with &, the shell prints a line which looks like:

[1] 1234

indicating that the job that was started asynchronously was job number 1 and had one (top-level) process whose process ID was 1234.

There are several ways to refer to jobs in the shell. The character % introduces a job name. If you wish to refer to job number 1, you can name it as **%1**. Just naming a job brings it to the foreground; thus **%1** is a synonym for **fg %1**, bringing job 1 back into the foreground.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a + and the previous job with a –. The abbreviation %+ refers to the current job and %– refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), %% is also a synonym for the current job.

Status reporting

After each command completes, the shell checks if any child processes' states have changed. This is done so that it does not otherwise disturb your work. It normally informs you whenever a job's status changes but only just before it prints a prompt. **The immediate notification available in other versions of the shell is not supported under SPU OS.** Hence, the shell variable *notify* and the shell command *notify* are not supported.

Substitutions

The various transformations the shell performs on the input are described in the sections below in the order in which they occur.

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character ! and may begin *anywhere* in the input stream (with the provison that they *do not* nest) This ! may be preceded by an \ to prevent its special meaning; for convenience, a ! is passed unchanged when it is followed by a blank, tab, newline, =, or (. (History substitutions also occur when an input line begins with ↑. This special abbreviation will be described later.) Any input line that contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands that are input from the terminal and that consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. Commands are numbered sequentially from 1. The size of the history list is controlled by the *history* variable. The previous command is always retained, regardless of its value.

For definiteness, consider the following output from the *history* command:

```
 9 write michael
10 ex write.c
11 cat oldwrite.c
12 diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing ! in the prompt string.

With the current event (13) we can refer to previous events by event number **!11**, relatively (as in **!-2**, referring to the same event), by a prefix of a command word (as in **!d** for event 12, or **!wri** for event 9), or by a string contained in a word in the command (as in **!?mic?**, also referring to event 9). These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case, !! refers to the previous command; thus !! alone is essentially a *redo*.

To select words from an event, follow the event specification by : and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	*n*'th argument
↑	first argument, i.e. 1
$	last argument
%	word matched by (immediately preceding) ?*s*? search
x–*y*	range of words
–*y*	abbreviates 0–*y*
*	abbreviates ↑–$, or nothing if only 1 word in event
*x**	abbreviates *x*–$
x–	like *x** but omitting word $

The : separating the event specification from the word designator can be omitted if the argument selector begins with ↑, $, *, –, or %. A sequence of modifiers, each preceded by :, can be placed after the optional word designator. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing ".xxx" component, leaving the root name.
e	Remove all but the extension ".xxx" part.
s /*l*/*r*/	Substitute *r* for *l*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g., **g&**.
p	Print the new command, but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like **q**, but break into words at blanks, tabs, and newlines.

Unless preceded by a **g**, the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left-hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of /; a \ quotes the delimiter into the *l* and *r* strings. The character & in the right-hand side is replaced by the text from the left. A \ quotes & also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in **!?***s*?. The trailing delimiter in the substitution may be omitted if a newline follows immediately, as may the trailing ? in a contextual scan.

A history reference may be given without an event specification, e.g., !$. In this case, the reference is to the previous command. However, if a previous history reference occurred on the same line, this form repeats the previous reference. Thus **!?foo?↑ !$** gives the first and last arguments from the command matching **?foo?**.

The special history reference **!#** does not refer to any previous command, but refers to the current command. It can be used to duplicate any or all of the words just typed. For example, **ls /usr/src/lib !#^/libc** is equivalent to **ls /usr/src/lib /usr/src/lib/libc**.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a ↑. This is equivalent to **!:s↑** providing a convenient shorthand for substitutions on the text of the previous line. Thus, **↑lb↑lib** fixes the spelling of **lib** in the previous command.

Finally, a history substitution may be surrounded with { and }, if necessary, to insulate it from the characters that follow. Thus, after **ls –ld ~paul**, you might enter **!{l}a** to do **ls –ld ~paula**, while **!la** would look for a command starting with **la**.

Quotations with ' and "

The quotation of strings by ' and " can be used to prevent all or some of the remaining substitutions. Strings enclosed in ' are prevented any further interpretation. Strings enclosed in " may be expanded as described below.

In both cases, the resulting text becomes all or part of a single word; only in one special case (see **Command Substitution** below) does a " quoted string yield parts of more than one word; ' quoted strings never do.

Alias substitution

The shell maintains a list of aliases that can be established, displayed, and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands, and the first word of each command, left-to-right, is checked to see if it is an alias. If it is, the alias text is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus, if the alias for **ls –l** is **ls**, the command **ls /usr** would map to **ls –l /usr**, the argument list here being undisturbed. Similarly, if **lookup** was an alias for **grep \!↑ /etc/passwd**, then **lookup bill** would map to **grep bill /etc/passwd**.

If an alias is found, the word transformation of the input text is performed and the process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus, we can **alias print pr \!* | lpr** to make a command that *pr*'s its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle that causes command input to be echoed. The setting of this variable results from the –v command line option.

Other operations treat variables numerically. The @ command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as zero or more strings. For the purpose of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by $ characters. This expansion can be prevented by preceding the $ with a \ except within double quotation marks where it *always* occurs, and within single quotation marks where it *never* occurs. Strings quoted by ` are interpreted later (see **Command substitution** below) so $ substitution does not occur there until later, if at all. A $ is passed unchanged if

followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first command word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in " or given the **:q** modifier, the results of variable substitution may eventually be command and filename substituted. Within ", a variable whose value consists of multiple words expands to a portion of a single word, with the words of the variables value separated by blanks. When the **:q** modifier is applied to a substitution, the variable expands to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable that is not set.

$name
${name}
: Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters that would otherwise be part of it. Shell variables have names consisting of up to 20 letters, and digits starting with a letter. The underscore character is considered a letter. If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}
: May be used to select only some of the words from the value of *name*. The selector is subjected to $ substitution and may consist of a single number or two numbers separated by a –. The first word of a variable's value is numbered **1**. If the first number of a range is omitted, it defaults to **1**. If the last member of a range is omitted, it defaults to **$#name**. The selector * selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}
: Gives the number of words in the variable. This is useful for later use in a [selector].

$0
: Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}
: Equivalent to **$argv**[number].

$*
: Equivalent to **$argv**[*].

The modifiers **:h**, **:t**, **:r**, **:q**, and **:x** may be applied to the substitutions above, as may **:gh**, **:gt**, and **:gr**. If braces { } appear in the command form, then the modifiers must appear within the braces. *The current implementation allows only one : modifier on each $ expansion.*

The following substitutions may not be modified with : modifiers.

$?name
${?name}
: Substitutes the string **1** if name is set, **0** if it is not.

$?0

Substitutes **1** if the current input filename is known, **0** if it is not.

$$

Substitutes the decimal process number of the parent shell.

$<

Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and filename substitution

The remaining substitutions, command and filename substitutions, are applied selectively to the arguments of builtin commands. This means that portions of expressions that are not evaluated are not subjected to these expansions. For commands that are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in single quotation marks. The output from such a command is normally broken into separate words at blanks, tabs, and newlines, with null words being discarded; this text then replaces the original string. Within souble quotation marks, only newlines force new words; blanks and tabs are preserved.

In any case, the single, final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters *, ?, [, {, or begins with the character ~, then that word is a candidate for filename substitution, also known as **globbing**. This word is then regarded as a pattern and replaced with an alphabetically sorted list of filenames that match the pattern. In a list of words specifying filename substitution, it is an error for no pattern to match an existing filename, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching, the characters ~ and { are more akin to abbreviations.

In matching filenames, the . at the beginning of a filename or immediately following a /, as well as the /, must be matched explicitly. The * matches any string of characters, including the null string. The ? matches any single character. The sequence [...] matches any one of the characters enclosed. Within [...], a pair of characters separated by – matches any character lexically between the two.

The ~ at the beginning of a filename is used to refer to home directories. Standing alone i.e., ~, it expands to the invoker's home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits, and – characters, the shell searches for a user with that name and substitutes the user's home directory. Thus, **~ken** might expand to **/usr/ken** and **~ken/chmach** to **/usr/ken/chmach**. If the ~ is followed by a character other than a letter or /, or appears other than at the beginning of a word, it is left undisturbed.

The metanotation **a{b,c,d}e** is shorthand for **abe ace ade**. Left-to-right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus, **~source/s1/{oldls,ls}.c** expands to **/usr/source/s1/oldls.c /usr/source/s1/ls.c** (whether or not these files exist) without any chance of error, if the home directory for **source** is **/usr/source**. Similarly, **../{memo,*box}** might expand to **../memo ../box ../mbox**. (Note that **memo** was not sorted with the results of matching ***box**.) As a special case, **{**, **}**, and **{}** are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name
: Open file *name* (which is first variable, command, and filename expanded) as the standard input.

<< word
: Read the shell input up to a line that is identical to *word. word* is not subjected to variable, filename, or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ', or ` appears in *word,* variable and command substitution is performed on the intervening lines, allowing \ to quote $, \, and `. Commands that are substituted have all blanks, tabs, and newlines preserved, except for the final newline, which is dropped. The resultant text is placed in an anonymous temporary file that is given to the command as standard input.

> name
>! name
>& name
>&! name
: The file *name* is used as standard output. If the file does not exist, then it is created; if the file exists, it is truncated and its previous contents are lost.

  If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g., a terminal or **/dev/null**) or an error results. This helps prevent accidental destruction of files. In this case, the ! forms can be used to suppress this check.

  The forms involving & route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way that < input filenames are.

>> name
>>& name
>>! name
>>&! name
: Uses file *name* as standard output, like >, but places output at the end of the file. If the variable *noclobber* is set, then it is an error for the file not to exist unless one of the ! forms is given. Otherwise similar to >.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands that run from a file of shell commands have no access to the text of the commands by default; rather, they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block-read its input. Note that the default standard input for a command run detached is **not** modified to be the empty file **/dev/null**; rather, the standard input remains as the original standard input of the shell. If this is a terminal, and if the process attempts to read from the terminal, then the process will block and the user will be notified (see **Jobs** above)

Diagnostic output may be directed through a pipe with the standard output. Simply use the form |& rather than just |.

Expressions

A number of the builtin commands, to be described subsequently, take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the *@, exit, if,* and *while* commands. The following operators are available:

|| && | ↑ & == != =~ !~ <= >= < > << >> + - * / % ! ~ ( )

Here, the precedence increases to the right: ==, !=, =~, and !~, <=, >=, <, and >, << and >>, + and −, *, /, and % being, in groups, at the same level. The ==, !=, =~, and !~ operators compare their arguments as strings; all others operate on numbers. The operators =~ is pattern equivalence and !~ is pattern non-equivalence much as != is numerical non-equivalence and == is numerical equivalence. Pattern (non)equivalence means the right side of the expression is a *pattern* (containing, e.g., *, ?, and instances of [...]) against which the left operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings that begin with **0** are considered octal numbers. Null or missing arguments are considered **0**. The result of all expressions are strings that represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions that are syntactically significant to the parser (&, |, <, >, (, and)), they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in { and } and file enquiries of the form **−l name**, where **l** is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded, and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all enquiries return false, i.e. **0**. Command executions succeed, returning true, i.e. **1**, if the command exits with status **0**; otherwise they fail, returning false, i.e., **0**. If more detailed status information is required, then the command should be executed outside of an expression and the variable *status* examined.

Control flow

The shell contains a number of commands that can be used to regulate the flow of control in command files, shell scripts, and, in limited but useful ways, from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if-then-else* form of the *if* statement, require that the major keywords appear in a single, simple command on an input line, as shown below.

If the shells input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward gotos will succeed on nonseekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last, it is executed in a subshell.

alias
alias *name*
alias *name wordlist*
The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified *wordlist* as the alias of *name; wordlist* is command and filename

substituted. *Name* is not allowed to be *alias* or *unalias*.

alloc
Shows the amount of dynamic core in use, broken down into used and free core, and the address of the last location in the heap. With an argument, it shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

break
Causes execution to resume after the *end* of the nearest enclosing *foreach* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a *switch*, resuming after the *endsw*.

case *label:*
A label in a *switch* statement as discussed below.

cd
cd *name*
chdir
chdir *name*
Change the shell's working directory to directory *name*. If no argument is given, then change to the home directory of the user. If *name* is not found as a subdirectory of the current directory (and does not begin with /, ./, or ../), then each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails, but *name* is a shell variable whose value begins with /, then this is tried to see if it is a directory.

continue
Continue execution of the nearest enclosing *while* or *foreach*. The remaining commands on the current line are executed.

default:
Labels the default case in a *switch* statement. The default should come after all *case* labels.

dirs
Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.

echo *wordlist*
echo **–n** *wordlist*
The specified words are written to the shells standard output, separated by spaces, and terminated with a newline, unless the **–n** option is specified.

else
end
endif
endsw
See the description of the *foreach*, *if*, *switch*, and *while* statements below.

eval *arg* ...
(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See *tset*(1) for an example of using *eval*.

exec *command*
The specified command is executed in place of the current shell.

exit
exit *(expr)*
The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

fg
fg *%job ...*
Brings the current or specified jobs into the foreground.

foreach *name (wordlist)*
...
end
The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching *end* are executed. (Both *foreach* and *end* must appear alone on separate lines.)

The builtin command *continue* may be used to continue the loop prematurely and the builtin command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with **?** before any statements in the loop are executed. If you make a mistake entering in a loop at the terminal, you can rub it out.

glob *wordlist*
glob is like *echo*, but no \ escapes are recognized and words are delimited by null characters in the output. This is useful for programs that wish to use the shell to filename expand a list of words.

goto word
The specified *word* is filename and command expanded to yield a string of the form **label**. The shell rewinds its input as much as possible and searches for a line of the form **label:**, possibly preceded by blanks or tabs. Execution continues after the specified line.

hashstat
Prints a statistics line indicating how effective the internal hash table has been at locating commands and avoiding *exec*s. An *exec* is attempted for each component of the *path*, where the hash function indicates a possible hit, and in each component that does not begin with a */*.

history
history *n*
history **–r** *n*
history **–h** *n*
Displays the history event list; if *n* is given, only the *n* most recent events are printed. The **–r** option reverses the order of printout to be most recent first. The **–h** option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourcing using the **–h** option to *source*.

if *(expr) command*
If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the remaining *if* command. *Command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is *not* executed (this is a bug).

if *(expr)* **then**
...
else if *(expr2)* **then**
...

else
...
endif

If the specified *expr* is true, then the commands to the first *else* are executed; **else if** *expr2* is true then the commands to the second else are executed, etc. Any number of pairs are possible; only one **endif** is needed. The **else** part is likewise optional. (The words **else** and **endif** must appear at the beginning of input lines; the **if** must appear alone on its input line or after an **else.**)

jobs
jobs –l

Lists the active jobs; given the –l options lists process ID's in addition to the normal information.

kill *%job*
kill *–sig %job ...*
kill *pid*
kill *–sig pid ...*
kill –l

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in */usr/include/signal.h,* stripped of the prefix **SIG**). The signal names are listed by **kill –l**. There is no default, entering **kill** does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process is sent a CONT (continue) signal as well.

The terminate signal kills processes that do not catch the signal; **kill –9 ...** is a sure kill, as the KILL (9) signal cannot be caught. The killed process must belong to the current user unless the user is the super user.

login

Terminate a login shell, replacing it with an instance of **/bin/login.** This is one way to log off (included for compatibility with *sh*(1)).

logout

Terminate a login shell. Especially useful if *ignoreeof* is set.

nohup
nohup *command*

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with **&** are effectively *nohup*ed.

onintr
onintr –
onintr *label*

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts that are to terminate shell scripts or to return to the terminal command input level. The second form **onintr –** causes all interrupts to be ignored. The final form causes the shell to execute a **goto label** when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning, and interrupts continue to be ignored by the shell and all invoked commands.

popd

popd *+n*

Pops the directory stack, returning to the new top directory. With an argument *+n* discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd
pushd *name*
pushd +n

With no arguments, *pushd* exchanges the top two elements of the directory stack. Given a *name* argument, *pushd* pushes the current working directory (as in *cwd*) onto the directory stack, then changes to the new directory (ala *cd*). With a numeric argument, pushd rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

rehash

Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories or if a systems programmer changes the contents of one of the system directories.

repeat *count command*

The specified *command,* which is subject to the same restrictions as the *command* in the one-line *if* statement above, is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

set
set *name*
set *name=word*
set *name[index]=word*
set *name=(wordlist)*

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word.* The fourth form sets the *index*th component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist.* In all cases, the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv *name value*

Sets the value of environment variable *name* to be *value,* a single string. The most commonly used environment variable USER, TERM, and PATH are automatically imported to and exported from the *csh* variables *user, term,* and *path*; there is no need to use *setenv* for these.

shift
shift *variable*

The members of *argv* are shifted to the left, discarding *argv[1].* It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name
source **–h** *name*

The shell reads commands from *name. Source* commands may be nested; if they are nested too deeply, the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Normally, input during *source* commands is not placed on the history list; the **–h** option causes the commands to be placed in the history

list without being executed. Note that since *source* is a *csh* builtin function, ^Z and ^C will not work in it. Attempting to stop or suspend the job will result in an attempt to stop or suspend the shell.

switch *(string)*
case *str1:*
...
breaksw
...
default:
...
breaksw
endsw

Each case label is successively matched against the specified *string* (which is first command and filename expanded). The file metacharacters *, ?, and [...] may be used in the case labels, which are variable expanded. If none of the labels match before a **default** label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise, control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

umask
umask *value*

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002, giving all access to the group and read and execute access to others, or 022, giving yourself all access, but no write access for users in the group or others.

unalias *pattern*

All aliases whose names match the specified pattern are discarded. Thus, all aliases are removed by **unalias** *. It is not an error if nothing is *unaliased*.

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unset *pattern*

All variables whose names match the specified pattern are removed. Thus, all variables are removed by **unset** *; this has noticeably distasteful side effects. It is not an error for nothing to be *unset*.

unsetenv *pattern*

Removes all variables whose names match the specified pattern from the environment. See the *setenv* command above and *printenv*(1).

wait

All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

whence *name ...*
whence **-w** *name ...*

Searches aliases, builtins, and finally the user's *path* for the program file that would be executed if *name* were to be typed as a command. The output can be in one of three forms: the full path of the command that would be executed, an indication of the wordlist that *name* is aliased to, or notification that the command in question is built in to the shell. If no output is given, the command was not found anywhere in the user's *path*, as a C shell builtin, or as an alias. *whence* may be invoked with multiple names, and each one will be processed in order. If the second form is used, the output will be in the same style as

which(1). The first form is perhaps more complete since it reports on C shell builtin commands, but both forms run much faster than the *which*(1) utility.

Examples:

```
% whence which
/bin/which
% alias h history
% whence h
alias/h 'history'
% whence history
builtin/history
```

while *(expr)*
...
end

While the specified expression evaluates nonzero, the commands between the *while* and the matching end are evaluated. *break* and *continue* may be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here, for the first time, through the loop as for the *foreach* statement if the input is a terminal.

@
@ *name = expr*
@ *name[index] = expr*

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains <, >, &, or |, then at least part of the expression must be placed within ( ). The third form assigns the value of *expr* to the *index*th argument of *name*. Both *name* and its *index*th component must already exist.

The operators *=, +=, etc., are available as in C. The space separating the name from the assignment operator is optional. Spaces, however, are mandatory in separating components of *expr* that would otherwise be single words.

Special postfix ++ and -- operators increment and decrement *name* respectively, i.e., **@i++**.

Predefined and environment variables

The following variables have special meaning to the shell. Of these, *argv, cwd, home, path, prompt, shell,* and *status* are always set by the shell. Except for *cwd* and *status,* this setting occurs only at initialization; these variables are not modified unless this is done explicitly by the user.

This shell copies the environment variable USER into the variable *user,* TERM into *term,* and HOME into *home,* and copies these back into the environment whenever the normal shell variables are reset. The environment variable PATH is likewise handled. It is not necessary to worry about its setting other than in the file *.cshrc,* as inferior *csh* processes import the definition of *path* from the environment and re-export it if you change it.

argv Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e., **$1** is replaced by **$argv[1]**, etc.

cdpath Gives a list of alternate directories searched to find subdirectories in *chdir* commands.

cwd The full pathname of the current directory.

echo Set when the **-x** command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands,

all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are done selectively.

histchars Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character !. The second character of its value replaces the character ↑ in quick substitutions.

history Can be given a numeric value to control the size of the history list. Any command that has been referenced in this many events is not discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.

home The home directory of the invoker, initialized from the environment. The filename expansion of ~ refers to this variable.

ignoreeof If set, the shell ignores end-of-file from input devices that are terminals. This prevents shells from accidentally being killed by **CTRL-D**'s.

mail The files where the shell checks for mail. This is done after each command completion that results in a prompt, if a specified interval has elapsed. The shell says **You have new mail**, if the file exists with an access time not greater than its modify time.

If the first word of the value of *mail* is numeric, it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says **New mail in** *name* when there is mail in the file *name*.

• **noclobber** As described in the section on **Input/output**, restrictions are placed on output redirection to ensure that files are not accidentally destroyed and that >> redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts that are not dealing with filenames or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. **echo** [still gives an error.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable, then only full pathnames will execute. The usual search path is ., */bin*, and */usr/bin*, but this may vary from system to system. For the superuser, the default search path is */etc*, */bin*, and */usr/bin*. A shell which is given neither the **–c** nor the **–t** option normally hashes the contents of the directories in the *path* variable after reading *.cshrc* and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* builtin, or the commands may not be found.

prompt The string that is printed before each command is read from an interactive terminal input. If a ! appears in the string, it is replaced by the current event number unless a preceding \ is given. Default is %, or # for the superuser.

savehist is given a numeric value to control the number of entries of the history list that are saved in ~*/.history* when the user logs out. Any command that has been referenced in this many events will be saved. During start-up, the shell sources ~*/.history* into the history list, enabling history to be saved across logins.

Values of *savehist* that are too large will slow down the shell during start up.

shell The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set, but that are not executable by the system. (See the description of **Non-builtin Command Execution** below.) *Shell* is initialized to the system-dependent home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands that fail return exit status **1**; all other builtin commands set status **0**.

verbose Set by the **–v** command line option, this causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is not a builtin command, the shell attempts to execute the command via *execve*(2). Each word in the variable *path* names a directory from which the shell attempts to execute the command. If it is given neither a **–c** nor a **–t** option, the shell hashes the names in these directories into an internal table so that it will only try an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash*), or if the shell was given a **–c** or **–t** argument, and in any case for each directory component of *path* that does not begin with a /, the shell concatenates with the given command name to form a pathname of a file that it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus **(cd ; pwd) ; pwd** prints the *home* directory; leaving you where you were (printing this after the home directory), while **cd ; pwd** leaves you in the *home* directory. Parenthesized commands are most often used to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands, and a new shell is spawned to read it.

If there is an *alias* for *shell*, then the words of the alias are prepended to the argument list to form the shell command. The first word of the *alias* should be the full pathname of the shell (e.g., **$shell**). Note that this is a special, late occurring, case of *alias* substitution and only allows words to be prepended to the argument list without modification.

Argument list processing

If argument 0 to the shell is –, then this is a login shell. The flag arguments are interpreted as follows:

–c Commands are read from the single following argument which must be present. Any remaining arguments are placed in *argv*.

–e The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

–f The shell starts faster because it neither searches for nor executes commands from the file *.cshrc* in the invoker's home directory.

–i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

–n Commands are parsed, but not executed. This aids in syntactic checking of shell scripts.

–s Command input is taken from the standard input.

–t A single line of input is read and executed. A \ may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the *verbose* variable to be set with the effect that command input is echoed after history substitution.

-x Causes the *echo* variable to be set so that commands are echoed immediately before execution.

-V Causes the *verbose* variable to be set even before *.cshrc* is executed.

-X Is to **-x** as **-V** is to **-v**.

After processing of flag arguments, if arguments remain but none of the **-c, -i, -s,** or **-t** options were given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file and saves its name for possible resubstitution by **$0**. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a **standard** shell if the first character of a script is not a #, i.e., if the script does not start with a comment. Remaining arguments initialize the variable *argv*.

Signal handling

The shell normally ignores *quit* signals. Jobs running detached by & are immune to signals generated from the keyboard, including hangups. Other signals have the values that the shell inherited from its parent. The shells handling of interrupts and terminate signals in shell scripts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise, this signal is passed on to children from the state in the shells parent. In no case are interrupts allowed when a login shell is reading the file *.logout*.

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell after */etc/login* at login.
~/.logout	Read by login shell at logout.
/bin/sh	Standard shell for shell scripts not starting with a #.
/tmp/sh*	Temporary file for <<.
/etc/login	Read by login shell after *.cshrc* at login.
/etc/logout	Read by login shell after *.logout* at logout.
/etc/passwd	Source of home directories for ~**name**.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command, which involves filename expansion, is limited to 1/6 the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

SEE ALSO

sh(1), access(2), execve(2), fork(2), pipe(2), sigvec(2), umask(2), wait(2), tty(4), b.out(5)

"An Introduction to the C Shell" in the *ConvexOS Tutorial Papers*

BUGS

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by ?, are not placed in the *history* list. Control structure should be parsed rather than being recognized as builtin commands. This would allow control commands to be placed anywhere, to be combined with |, and to be used with & and ; metasyntax.

It should be possible to use the : modifiers on the output of command substitutions. All and more than one : modifier should be allowed on $ substitutions.

NAME

date – print and set the date

SYNOPSIS

date [-u] [-z zone] [yymmddhhmm [.ss]]

DESCRIPTION

If no arguments are given, the current date and time are printed. If a date is specified, the current date is set. *yy* is the last two digits of the year; the first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *.ss* is optional and is the seconds. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The year, month and day may be omitted, the current values being the defaults.

The system operates in GMT. **Date** takes care of the conversion to and from local standard and daylight time. The *-u* flag is used to display the date in GMT (universal) time. This flag may also be used to set GMT time.

The *-z* flag is used to set the time zone where *zone* is a string which identifies the local time zone. The string can be in one of two forms. The simplest form of the string is a mnemonic for the time zone as follows:

Zone Name	Time Zone
ast | adt	US: Atlantic
est | edt	US: Eastern
cst | cdt	US: Central
mst | mdt	US: Mountain
pst | pdt	US: Pacific
eet | eetdst	Eastern European
met | metdst	Middle European
wet | wetdst	Western European
aest | aedt	Australia: Eastern
acst | acdt	Australia: Central
awst | awdt	Australia: Western

The more general form of the string is

[+|-]hh:mm[,dst_rule_name]

where *hh:mm* is the number of hours and minutes that the local time zone is east *(-)* or west *(+)* of GMT. The optional *dst_rule_name* specifies the daylight savings time rule to be used. If *dst_rule_name* is not specified, then it is assumed that daylight savings time does not apply to the local time zone. The following *dst_rule_name* mnemonics may be used:

Dst_rule_name	DST Rule
us	US
eet	Eastern European
met	Middle European
wet	Western European
aus	Australia

Note that the time zone must be set each time that UNIX is booted. The time zone mnemonic displayed by **date** is determined by the daylight savings time rules used in **ctime(3)**. Hence, specifying the time zone to be *cdt* in January will result in CST being displayed by **date** since daylight savings time is not in affect in January for the central time zone in the United States.

The time zone mnemonics displayed by **date** can be modified by specifying the desired mnemonics in the environmental variable TZNAME. For example,

```
TZNAME="STD,DST"; export TZNAME
```

will result in STD and DST being displayed as the standard time and daylight savings time mnemonics, respectively.

FILES

/usr/adm/wtmp to record time-setting

SEE ALSO

ctime(3)
timezone(3)
utmp(5)

DIAGNOSTICS

'Failed to set date: Not owner' or 'Failed to set time zone: Not owner' if you try to change the date but are not the super-user.

NAME

dd – convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

option	*values*
if=	input file name; standard input is default
of=	output file name; standard output is default
ibs=*n*	input block size *n* bytes (default 512)
obs=*n*	output block size (default 512)
bs=*n*	set both input and output block size, superseding *ibs* and *obs;* also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
files=*n*	copy *n* files from (tape) input
seek=*n*	seek *n* records from beginning of output file before copying
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to *ibs*
... , ...	several comma-separated conversions

Where sizes are specified, a number of bytes is expected. A number may end with **k, b** or **w** to specify multiplication by 1024, 512, or 2 respectively; a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and new-line added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size *cbs*.

After completion, **dd** reports the number of whole and partial input and output blocks.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x*:

```
dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. **Dd** is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

To skip over a file before copying from magnetic tape do

```
(dd of=/dev/null; dd of=x) </dev/rmt0
```

SEE ALSO

cp(1)

DIAGNOSTICS

f+p records in(out): numbers of full and partial records read(written)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. The 'ibm' conversion, while less blessed as a standard, corresponds better to certain IBM print train conventions. There is no universal solution.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

NAME

df – disk free

SYNOPSIS

df [filesystem] ...

DESCRIPTION

Df prints out the number of free blocks available on the *filesystems*. If no file system is specified, the free space on all of the normally mounted file systems is printed.

FILES

Default file systems vary with installation.

NAME

echo – echo arguments

SYNOPSIS

echo [**–n**] [arg] ...

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a newline on the standard output. If the flag **–n** is used, no newline is added to the output.

Echo is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send diagnostics to the standard error file, do 'echo ... 1>&2'.

NAME

grep – search a file for a pattern

SYNOPSIS

grep [option] ... expression [file] ...

DESCRIPTION

Grep searches the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular expressions in the style of *xed*(1); it uses a compact nondeterministic algorithm. The following options are recognized.

–v All lines but those matching are printed.

–c Only a count of matching lines is printed.

–l The names of files with matching lines are listed (once) separated by newlines.

–n Each line is preceded by its relative line number in the file.

–b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

–i The case of letters is ignored in making comparisons — that is, upper and lower case are considered identical.

–y Equivalent to the **–i** switch.

–s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

–e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a –.

In all cases the file name is shown if there is more than one input file. Care should be taken when using the characters $ * [^ | ( ) and \ in the *expression* as they are also meaningful to the Shell. It is safest to enclose the entire *expression* argument in single quotes ' '.

In the following description 'character' excludes newline:

A \ followed by a single character other than newline matches that character.

The character ^ matches the beginning of a line.

The character $ matches the end of a line.

A . (period) matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in 'a–z0–9'. A] may occur only as the first character of the string. A literal – must be placed where it can't be mistaken as a range indicator.

A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression. A regular expression followed by a + (plus) matches a sequence of 1 or more matches of the regular expression. A regular expression followed by a ? (question mark) matches a sequence of 0 or 1 matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

Unlike CONVEX UNIX, there is only one *grep* in SPU UNIX, but that's only because we're tightwads about SPU disk space.

SEE ALSO

sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

BUGS

Lines are limited to 256 characters; longer lines are truncated.

NAME

kill – terminate a process with extreme prejudice

SYNOPSIS

kill [–signo] processid ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. If a signal number preceded by '–' is given as first argument, that signal is sent instead of terminate (see **signal(2)**). This will kill processes that do not catch the signal; in particular 'kill –9 ...' is a sure kill.

By convention, if process number 0 is specified, all members in the process group (i.e. processes resulting from the current login) are signaled.

The killed processes must belong to the current user unless he is the super-user. To shut the system down and bring it up single user the super-user may use 'kill –1 1'; see **init(8)**.

The process number of an asynchronous process started with '&' is reported by the shell. Process numbers can also be found by using **ps(1)**.

SEE ALSO

ps(1)
kill(2)
signal(2)

NAME

less – opposite of more

SYNOPSIS

less -?

less [-[+]**aABcCdeEfgimMnNqQruUsw**] [-**b***N*] [-**h***N*] [-**x***N*] [-[**z**]*N*]
[-**P**[**mM**=]*string*] [-[**lL**]*logfile*] [-**k***keyfile*]
[+*cmd*] [-**t***tag*] [*filename*]...

DESCRIPTION

Less is a program similar to *more* (1), but which allows backwards movement in the file as well as forward movement. Also, *less* does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like *vi* (1). *Less* uses termcap (or terminfo on some systems), so it can run on a variety of terminals. There is even limited support for hard-copy terminals. (On a hardcopy terminal, lines which should be printed at the top of the screen are prefixed with an up-arrow.)

Commands are based on both *more* and *vi*. Commands may be preceded by a decimal number, called N in the descriptions below. The number is used by some commands, as indicated.

COMMANDS

In the following descriptions, ^X means control-X. ESC stands for the ESCAPE key; for example ESC-v means the two character sequence "ESCAPE", then "v".

h or H Help: display a summary of these commands. If you forget all the other commands, remember this one.

SPACE or ^V or f or ^F
Scroll forward N lines, default one window (see option -z below). If N is more than the screen size, only the final screenful is displayed. Warning: some systems use ^V as a special literalization character.

z Like SPACE, but if N is specified, it becomes the new window size.

RETURN or ^N or e or ^E or j or ^J
Scroll forward N lines, default 1. The entire N lines are displayed, even if N is more than the screen size.

d or ^D
Scroll forward N lines, default one half of the screen size. If N is specified, it becomes the new default for subsequent d and u commands.

b or ^B or ESC-v
Scroll backward N lines, default one window (see option -z below). If N is more than the screen size, only the final screenful is displayed.

w Like ESC-v, but if N is specified, it becomes the new window size.

y or ^Y or ^P or k or ^K
Scroll backward N lines, default 1. The entire N lines are displayed, even if N is more than the screen size. Warning: some systems use ^Y as a special job control character.

u or ^U
Scroll backward N lines, default one half of the screen size. If N is specified, it becomes the new default for subsequent d and u commands.

r or ^R or ^L
Repaint the screen.

R Repaint the screen, discarding any buffered input. Useful if the file is changing while it is being viewed.

g or < or ESC-<
Go to line N in the file, default 1 (beginning of file). (Warning: this may be slow if N is large.)

G or > or ESC->
Go to line N in the file, default the end of the file. (Warning: this may be slow if N is large, or if N is not specified and standard input, rather than a file, is being read.)

p Go to a position N percent into the file. N should be between 0 and 100. (This works if standard input is being read, but only if *less* has already read to the end of the file. It is always fast, but not always useful.)

% If a bracket appears in the top line displayed on the screen, the % command will go to the matching bracket. A "bracket" is one of the characters "{ } [ ] ( )".

{ If a left curly bracket appears in the top line displayed on the screen, the { command will go to the matching right curly bracket. If there is more than one left curly bracket on the top line, a number N may be used to specify the N-th bracket on the line.

} If a right curly bracket appears in the top line displayed on the screen, the } command will go to the matching left curly bracket. If there is more than one right curly bracket on the top line, a number N may be used to specify the N-th bracket on the line.

(Like {, but applies to parentheses rather than curly brackets.

) Like }, but applies to parentheses rather than curly brackets.

[Like {, but applies to square brackets rather than curly brackets.

] Like }, but applies to square brackets rather than curly brackets.

m Followed by any lowercase letter, marks the current position with that letter.

' (Single quote.) Followed by any lowercase letter, returns to the position which was previously marked with that letter. Followed by another single quote, returns to the position at which the last "large" movement command was executed. Marks are preserved when a new file is examined, so the ' command can be used to switch between input files.

^X^X Same as single quote.

/pattern
Search forward in the file for the N-th line containing the pattern. N defaults to 1. The pattern is a regular expression, as recognized by *ed*. The search starts at the second line displayed (but see the -a option, which changes this).

?pattern
Search backward in the file for the N-th line containing the pattern. The search starts at the line immediately before the top line displayed.

ESC-/pattern
Search the entire file for the N-th line containing the pattern. The search starts at the first line in the file, regardless of what is displayed or the setting of the -a option.

/!pattern
Like /, but the search is for the N-th line which does NOT contain the pattern.

?!pattern
Like ?, but the search is for the N-th line which does NOT contain the pattern.

ESC-/!pattern
Like ESC-/, but the search is for the N-th line which does NOT contain the pattern.

n Repeat previous search, for N-th line containing the last pattern (or NOT containing the last pattern, if the previous search was /! or ?!).

ESC-n Repeat previous search, but in the reverse direction.

E [filename]
Examine a new file. If the filename is missing, the "current" file (see the N and P commands below) from the list of files in the command line is re-examined. A percent sign (%) in the filename is replaced by the name of the current file. A pound sign (#) is replaced by the name of the previously examined file.

^X^V or :e
Same as E. Warning: some systems use ^V as a special literalization character.

N or :n Examine the next file (from the list of files given in the command line). If a number N is specified (not to be confused with the command N), the N-th next file is examined.

P or :p Examine the previous file. If a number N is specified, the N-th previous file is examined.

= or ^G or :f
Prints some information about the file being viewed, including its name and the line number and byte offset of the bottom line being displayed. If possible, it also prints the length of the file, the number of lines in the file and the percent of the file above the last displayed line.

- Followed by one of the command line option letters (see below), this will change the setting of that option and print a message describing the new setting. If the option letter has a numeric value (such as -b or -h), or a string value (such as -P or -t), a new value may be entered after the option letter.

_ (Underscore.) Followed by one of the command line option letters (see below), this will print a message describing the current setting of that option. The setting of the option is not changed.

+cmd Causes the specified cmd to be executed each time a new file is examined. For example, +G causes *less* to initially display each file starting at the end rather than the beginning.

V Prints the version number of *less* being run.

q or :q or :Q or ZZ
Exits *less*.

The following two commands may or may not be valid, depending on your particular installation.

v Invokes an editor to edit the current file being viewed. The editor is taken from the environment variable EDITOR, or defaults to "vi". See also the discussion of EDITPROTO under the section on PROMPTS below.

! shell-command
Invokes a shell to run the shell-command given. A percent sign (%) in the command is replaced by the name of the current file. A pound sign (#) is replaced by the name of the previously examined file. "!!" repeats the last shell command. "!" with no shell command simply invokes a shell. In all cases, the shell is taken from the environment variable SHELL, or defaults to "sh".

OPTIONS

Command line options are described below. Most options may be changed while *less* is running, via the "–" command.

Options are also taken from the environment variable "LESS". For example, to avoid typing "less -options ..." each time *less* is invoked, you might tell *csh:*

```
setenv LESS "-options"
```

or if you use *sh:*

LESS="-options"; export LESS

The environment variable is parsed before the command line, so command line options override the LESS environment variable. If an option appears in the LESS variable, it can be reset to its default on the command line by beginning the command line option with "-+".

A dollar sign ($) may be used to signal the end of an option string. This is important only for options like -P which take a following string.

-? This option displays a summary of the commands accepted by *less* (the same as the h command). If this option is given, all other options are ignored, and *less* exits after the help screen is viewed. (Depending on how your shell interprets the question mark, it may be necessary to quote the question mark, thus: "-\?".)

-a Normally, forward searches start just after the top displayed line (that is, at the second displayed line). Thus, forward searches include the currently displayed screen. The -a option causes forward searches to start just after the bottom line displayed, thus skipping the currently displayed screen.

-A The -A option causes searches to start at the second SCREEN line displayed, as opposed to the default which is to start at the second REAL line displayed. For example, suppose a long real line occupies the first three screen lines. The default search will start at the second real line (the fourth screen line), while the -A option will cause the search to start at the second screen line (in the midst of the first real line). (This option is rarely useful.)

-b The -b*n* option tells *less* to use a non-standard number of buffers. Buffers are 1K, and normally 10 buffers are used (except if data in coming from standard input; see the -B option). The number *n* specifies a different number of buffers to use.

-B Normally, when data is coming from standard input, buffers are allocated automatically as needed, to avoid loss of data. The -B option disables this feature, so that only the default number of buffers are used. If more data is read than will fit in the buffers, the oldest data is discarded.

-c Normally, *less* will repaint the screen by scrolling from the bottom of the screen. If the -c option is set, when *less* needs to change the entire display, it will paint from the top line down.

-C The -C option is like -c, but the screen is cleared before it is repainted.

-d Normally, *less* will complain if the terminal is dumb; that is, lacks some important capability, such as the ability to clear the screen or scroll backwards. The -d option suppresses this complaint (but does not otherwise change the behavior of the program on a dumb terminal).

-e Normally the only way to exit *less* is via the "q" command. The -e option tells *less* to automatically exit the second time it reaches end-of-file.

-E The -E flag causes *less* to exit the first time it reaches end-of-file.

-f Normally, *less* will refuse to open a non-regular file (that is, a file which is a directory or a device special file). The -f flag forces *less* to open such files.

-g Force input characters to 7 bits (that is, mask off the high order bit). The default is to use full 8 bit characters.

-h Normally, *less* will scroll backwards when backwards movement is necessary. The -h option specifies a maximum number of lines to scroll backwards. If it is necessary to move backwards more than this many lines, the screen is repainted in a forward

direction. (If the terminal does not have the ability to scroll backwards, -h0 is implied.)

-i The -i option causes searches to ignore case; that is, uppercase and lowercase are considered identical. Also, text which is overstruck or underlined can be searched for.

-k The -k option, followed immediately by a filename, will cause *less* to open and interpret the file as a *lesskey* (1) file. Multiple -k options may be specified. If a file called .less exists in the user's home directory, this file is also used as a *lesskey* file.

-l The -l option, followed immediately by a filename, will cause *less* to copy its input to the named file as it is being viewed. This applies only when the input file is a pipe, not an ordinary file. If the file already exists, *less* will ask for confirmation before overwriting it.

-L The -L option is like -l, but it will overwrite an existing file without asking for confirmation.

If no log file has been specified, the -l and -L options can be used from within *less* to specify a log file. Without a file name, they will simply report the name of the log file. The "s" command is equivalent to specifying -l from within *less*.

-m Normally, *less* prompts with a colon. The -m option causes *less* to prompt verbosely (like *more),* with the percent into the file.

-M The -M option causes *less* to prompt even more verbosely than *more.*

-n The -n flag suppresses line numbers. The default (to use line numbers) may cause *less* to run more slowly in some cases, especially with a very large input file. Suppressing line numbers with the -n flag will avoid this problem. Using line numbers means: the line number will be displayed in the verbose prompt and in the = command, and the v command will pass the current line number to the editor (see also the discussion of EDITPROTO in PROMPTS below).

-N The -N flag causes a line number to be displayed at the beginning of each line in the display.

-P The -P option provides a way to tailor the three prompt styles to your own preference. You would normally put this option in your LESS environment variable, rather than type it in with each *less* command. Such an option must either be the last option in the LESS variable, or be terminated by a dollar sign. -P followed by a string changes the default (short) prompt to that string. -Pm changes the medium (-m) prompt to the string, and -PM changes the long (-M) prompt. Also, -P= changes the message printed by the = command to the given string. All prompt strings consist of a sequence of letters and special escape sequences. See the section on PROMPTS for more details.

-q Normally, if an attempt is made to scroll past the end of the file or before the beginning of the file, the terminal bell is rung to indicate this fact. The -q option tells *less* not to ring the bell at such times. If the terminal has a "visual bell", it is used instead.

-Q Even if -q is given, *less* will ring the bell on certain other errors, such as typing an invalid character. The -Q option tells *less* to be quiet all the time; that is, never ring the terminal bell. If the terminal has a "visual bell", it is used instead.

-r Normally, control character are displayed using a carat and the equivalent non-control character. For example, a control-A (octal 001) is displayed as "^A". If the -r flag is set, "raw" control characters are displayed, without this translation. Warning: when this flag is used, *less* cannot keep track of the actual appearance of the screen (since this depends on how the screen responds to each type of control character). Thus, various display problems may result, such as long lines being split in the wrong place.

-s The -s option causes consecutive blank lines to be squeezed into a single blank line. This is useful when viewing *nroff* output.

-t The -t option, followed immediately by a TAG, will edit the file containing that tag. For this to work, there must be a file called "tags" in the current directory, which was previously built by the *ctags* (1) command. This option may also be specified from within *less* (using the – command) as a way of examining a new file. For *vi* compatibility, the command ":ta" is equivalent to specifying -t from within *less*.

-u If the -u option is given, backspaces are treated as printable characters; that is, they are sent to the terminal when they appear in the input.

-U If the -U option is given, backspaces are printed as the two character sequence "^H".

If neither -u nor -U is given, backspaces which appear adjacent to an underscore character are treated specially: the underlined text is displayed using the terminal's hardware underlining capability. Also, backspaces which appear between two identical characters are treated specially: the overstruck text is printed using the terminal's hardware boldface capability. Other backspaces are deleted, along with the preceding character.

-w Normally, *less* uses a tilde character to represent lines past the end of the file. The -w option causes blank lines to be used instead.

-x The -x*n* option sets tab stops every *n* positions. The default for *n* is 8.

-[z] When given a backwards or forwards window command, *less* will by default scroll backwards or forwards one screenful of lines. The -z*n* option changes the default scrolling window size to *n* lines. The z and w commands can also be used to change the window size. Note that the "z" in "-z*n*" is optional for compatibility with *more*.

+ If a command line option begins with +, the remainder of that option is taken to be an initial command to *less*. For example, +G tells *less* to start at the end of the file rather than the beginning, and +/xyz tells it to start at the first occurrence of "xyz" in the file. As a special case, +<number> acts like +<number>g; that is, it starts the display at the specified line number (however, see the caveat under the "g" command above). If the option starts with ++, the initial command applies to every file being viewed, not just the first one. The + command described previously may also be used to set (or change) an initial command for every file.

KEY BINDINGS

You may define your own *less* commands by using the program *lesskey* (1) to create a file called ".less" in your home directory. This file specifies a set of command keys and an action associated with each key. See the *lesskey* manual page for more details.

PROMPTS

The -P option allows you to tailor the prompt to your preference. The string given to the -P option replaces the specified prompt string. Certain characters in the string are interpreted specially. The prompt mechanism is rather complicated to provide flexibility, but the ordinary user need not understand the details of constructing personalized prompt strings.

A percent sign followed by a single character is expanded according to what the following character is:

%bX Replaced by the byte offset into the current input file. The b is followed by a single character (shown as X above) which specifies the line whose byte offset is to be used. If the character is a "t", the byte offset of the top line in the display is used, an "m" means use the middle line, a "b" means use the bottom line, and a "B" means use the line just after the bottom line.

%E Replaced by the name of the editor (from the EDITOR environment variable). See the

discussion of the EDITPROTO feature below.

%f Replaced by the name of the current input file.

%i Replaced by the index of the current file in the list of input files.

%lX Replaced by the line number of a line in the input file. The line to be used is determined by the X, as with the %b option.

%L Replaced by the line number of the last line in the input file.

%m Replaced by the total number of input files.

%pX Replaced by the percent into the current input file. The line used is determined by the X as with the %b option.

%s Replaced by the size of the current input file.

%t Causes any trailing spaces to be removed. Usually used at the end of the string, but may appear anywhere.

%x Replaced by the name of the next input file in the list.

If any item is unknown (for example, the file size if input is a pipe), a question mark is printed instead.

The format of the prompt string can be changed depending on certain conditions. A question mark followed by a single character acts like an "IF": depending on the following character, a condition is evaluated. If the condition is true, any characters following the question mark and condition character, up to a period, are included in the prompt. If the condition is false, such characters are not included. A colon appearing between the question mark and the period can be used to establish an "ELSE": any characters between the colon and the period are included in the string if and only if the IF condition is false. Condition characters (which follow a question mark) may be:

?a True if any characters have been included in the prompt so far.

?bX True if the byte offset of the specified line is known.

?e True if at end-of-file.

?f True if there is an input filename (that is, if input is not a pipe).

?lX True if the line number of the specified line is known.

?L True if the line number of the last line in the file is known.

?m True if there is more than one input file.

?n True if this is the first prompt in a new input file.

?pX True if the percent into the current input file of the specified line is known.

?s True if the size of current input file is known.

?x True if there is a next input file (that is, if the current input file is not the last one).

Any characters other than the special ones (question mark, colon, period, percent, and backslash) become literally part of the prompt. Any of the special characters may be included in the prompt literally by preceding it with a backslash.

Some examples:

?f%f:Standard input.

This prompt prints the filename, if known; otherwise the string "Standard input".

?f%f .?ltLine %lt:?pt%pt\%:?btByte %bt:-...

This prompt would print the filename, if known. The filename is followed by the line number, if known, otherwise the percent if known, otherwise the byte offset if known. Otherwise, a dash is printed. Notice how each question mark has a matching period, and how the % after the %pt is included literally by escaping it with a backslash.

?n?f%f .?m(file %i of %m) ..?e(END) ?x- Next\: %x..%t

This prints the filename if this is the first prompt in a file, followed by the "file N of N" message if there is more than one input file. Then, if we are at end-of-file, the string "(END)" is printed followed by the name of the next file, if there is one. Finally, any trailing spaces are truncated. This is the default prompt. For reference, here are the defaults for the other two prompts (-m and -M respectively). Each is broken into two lines here for readability only.

?n?f%f .?m(file %i of %m) ..?e(END) ?x- Next\: %x.:
?pB%pB\%:byte %bB?s/%s...%t

?f%f .?n?m(file %i of %m) ..?ltline %lt?L/%L. :byte %bB?s/%s. .
?e(END) ?x- Next\: %x.:?pB%pB\%..%t

And here is the default message produced by the = command:

?f%f .?m(file %i of %m) .?ltline %lt?L/%L. .
byte %bB?s/%s. ?e(END) :?pB%pB\%..%t

The prompt expansion features are also used for another purpose: if an environment variable EDITPROTO is defined, it is used as the command to be executed when the v command is invoked. The EDITPROTO string is expanded in the same way as the prompt strings. The default value for EDITPROTO is:

%E ?lm+%lm. %f

Note that this expands to the editor name, followed by a + and the line number, followed by the file name. If your editor loes not accept the "+linenumber" syntax, or has other differences in invocation syntax, the EDITPROTO variable can be changed to modify this default.

ENVIRONMENT VARIABLES

EDITOR
The name of the editor (used for the v command).

EDITPROTO
Editor prototype string (used for the v command). See discussion under PROMPTS.

HOME Name of the user's home directory (used to find a .less file).

LESS Flags which are passed to *less* automatically.

SHELL The shell used to execute the ! command, as well as to expand filenames.

TERM The type of terminal on which *less* is being run.

SEE ALSO

lesskey(1)

WARNINGS

The = command and prompts (unless changed by -P) report the line number of the line at the top of the screen, but the byte and percent of the line at the bottom of the screen.

NAME

ln - make a link

SYNOPSIS

ln name1 [name2]

DESCRIPTION

A *link* is a directory entry referring to a file; the same file (together with its size, all its protection information, etc.) may have several links to it. There is no way to distinguish a link to a file from its original directory entry; any changes in the file are effective independently of the name by which the file is known.

Ln creates a link to an existing file *name1*. If *name2* is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of *name1*.

It is forbidden to link to a directory or to link across file systems.

SEE ALSO

rm(1)

NAME

ls – list contents of directory

SYNOPSIS

ls [**–abcdfgilmqrstux1CFGRT**] name ...
l [*ls* options] name ...

DESCRIPTION

For each directory argument, **ls** lists the contents of the directory; for each file argument, **ls** repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The format chosen depends on whether the output is going to a teletype, and may also be controlled by option flags. The default format for a teletype is to list the contents of directories in multi-column format, with the entries sorted down the columns. (Files which are not the contents of a directory being interpreted are always sorted across the page rather than down the page in columns. This is because the individual file names may be arbitrarily long.) If the standard output is not a teletype, the default format is to list one entry per line. Finally, there is a stream output format in which files are listed across the page, separated by ',' characters. The **–m** flag enables this format; when invoked as *l*, this format is also used.

There are several options:

–l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

–t Sort by time modified (latest first) instead of by name, as is normal.

–a List all entries; usually '.' and '..' are suppressed.

–s Give size in blocks, including indirect blocks, for each entry.

–d If argument is a directory, list only its name, not its contents (mostly used with **–l** to get status on directory).

–r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

–u Use time of last access instead of last modification for sorting (**–t**) or printing (**–l**).

–c Use time of file creation for sorting or printing.

–i Print i-number in first column of the report for each file listed.

–f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **–l, –t, –s,** and **–r,** and turns on **–a;** the order is the order in which entries appear in the directory.

–g Give group ID instead of owner ID in long listing.

–m force stream output format

–1 force one entry per line output format, e.g. to a teletype

–C force multi-column output, e.g. to a file or a pipe

–q force printing of non-graphic characters in file names as the character '?'; this normally happens only if the output device is a teletype

–b force printing of non-graphic characters to be in the \ddd notation, in octal.

–x force columnar printing to be sorted across rather than down the page; this is the default if the last character of the name the program is invoked with is an 'x'.

-F cause directories to be marked with a trailing '/' and executable files to be marked with a trailing '*'; this is the default if the last character of the name the program is invoked with is a 'f'.

-G Gives the group ID in addition to the owner ID (cf. -g) in long listing.

-R recursively list subdirectories encountered.

-T force the printing of the directory being listed before its contents.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
m if the entry is a multiplexor-type character special file;
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise the user-execute permission character is given as **s** if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '-') is **t** if the 1000 bit of the mode is on. See **chmod(1)** for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for 'ls -l'.
/etc/group to get group ID's for 'ls -g'.

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "ls -s" is much different than "ls -s | lpr". On the other hand, not doing this setting would make old shell scripts which used **ls** almost certain losers.

Column widths choices are poor for terminals which can tab.

NAME

mkdir – make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777. Standard entries, ' . ', for the directory itself, and ' ' for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

rm(1)

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made. Otherwise it prints a diagnostic and returns nonzero.

NAME

more - file perusal filter for crt viewing

SYNOPSIS

more file

DESCRIPTION

More is a filter which allows examination of a continuous text one screenful at a time on a soft-copy terminal. It normally pauses after ecach screenful, printing <<< More (%%) >>> at the bottom of the screen. The user may then enter one of the following commands:

<CR>	Display next line.
<SP>	Display next screen.
t	Goto top of file.
q	Exit **more.**

If the standard output is not a soft-copy terminal, then **more** acts just like **cat(1)**. **More** can be terminated at any time via the interrupt key (normally **^C**).

FILES

/etc/termcap	Terminal data base

SEE ALSO

cat(1)

BUGS

Current version cannot be used as a filter.

NAME

mt – magnetic tape manipulating program

SYNOPSIS

mt [**-f** *tapename*] *command* [*count*]

DESCRIPTION

Mt is used to give commands to a magnetic tape drive. If *tapename* is not specified, the environment variable TAPE is used; if TAPE does not exist, *mt* uses the device */dev/rmt1*. Note that *tapename* must reference a raw (not block) tape device. By default *mt* performs the requested operation once. Operations may be performed multiple times by specifying *count*.

The available *command*(s) are listed below. Only as many characters as are required to uniquely identify a *command* need be specified.

eof, weof
Write *count* end-of-file marks at the current position on the tape.

fsf Forward space *count* files.

bsf Back space *count* files.

bsr Back space *count* records.

rewind
Rewind the tape (*Count* is ignored.)

retention
Retention the tape (*Count* is ignored.)

erase Erase the tape (*Count* is ignored.)

offline, rewoffl
Rewind the tape and place the tape unit offline (*Count* is ignored.)

status Print status information about the tape unit.

find Find data indicated by count, which should be a character string in this case. This command only applies to tapes in backup format (see backup(4)). Note that if string consists of multiple words, they must be quoted, such as:

```
mt find "/mnt partition"
```

Mt returns a 0 exit status if the operation(s) are successful, 1 if the command was unrecognized, and 2 if an operation failed.

RESTRICTIONS

mt fsr 1 cannot be used to skip over the tape mark between files.

Tape operations must be performed on *raw* tape devices not *block* tape devices.

FILES

*/dev/rmt** Raw magnetic tape interface

SEE ALSO

mtio(4), dd(1), ioctl(2), backup(5)

NAME

mv – move or rename files and directories

SYNOPSIS

mv file1 file2

mv file ... directory

DESCRIPTION

Mv moves (changes the name of) *file1* to *file2*.

If *file2* already exists, it is removed before *file1* is moved. If *file2* has a mode which forbids writing, **mv** prints the mode (see **chmod(2)**) and reads the standard input to obtain a line; if the line begins with **y**, the move takes place; if not, **mv** exits.

In the second form, one or more *files* are moved to the *directory* with their original file-names.

Mv refuses to move a file onto itself.

SEE ALSO

cp(1)
chmod(2)

BUGS

If *file1* and *file2* lie on different file systems, **mv** must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

Mv should take **–f** flag, like *rm*, to suppress the question if the target exists and is not writable.

NAME

od – octal, decimal, hex, ascii dump

SYNOPSIS

od [–format] [file] [[+]offset[.][**b**] [label]]

DESCRIPTION

Od displays *file*, or it's standard input, in one or more dump formats as selected by the first argument. If the first argument is missing, **–o** is the default. Dumping continues until end-of-file.

The meanings of the format argument characters are:

a Interpret bytes as characters and display them with their ACSII names. If the **p** character is given also, then bytes with even parity are underlined. The **P** character causes bytes with odd parity to be underlined. Otherwise the parity bit is ignored.

b Interpret bytes as unsigned octal.

c Interpret bytes as ASCII characters. Certain non-graphic characters appear as C escapes: null=\0, backspace=\b, formfeed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers. Bytes with the parity bit set are displayed in octal.

d Interpret (short) words as unsigned decimal.

h Interpret (short) words as unsigned hexadecimal.

i Interpret (short) words as signed decimal.

l Interpret long words as signed decimal.

o Interpret (short) words as unsigned octal.

s[n] Look for strings of ascii graphic characters, terminated with a null byte. *N* specifies the minimum length string to be recognized. By default, the minimum length is 3 characters.

v Show all data. By default, display lines that are identical to the last line shown are not output, but are indicated with an "*" in column 1.

w[n] Specifies the number of input bytes to be interpreted and displayed on each output line. If **w** is not specified, 16 bytes are read for each display line. If *n* is not specified, it defaults to 32.

x Interpret (short) words as hexadecimal.

An upper case format character implies the long or double precision form of the object.

The *offset* argument specifies the byte offset into the file where dumping is to commence. By default this argument is interpreted in octal. A different radix can be specified; If "." is appended to the argument, then *offset* is interpreted in decimal. If *offset* begins with "**x**" or "**0x**", it is interpreted in hexadecimal. If "**b**" ("**B**") is appended, the offset is interpreted as a block count, where a block is 512 (1024) bytes. If the *file* argument is omitted, an *offset* argument must be preceded by "+".

The radix of the displayed address will be the same as the radix of the *offset*, if specified; otherwise it will be octal.

Label will be interpreted as a pseudo-address for the first byte displayed. It will be shown in "()" following the file offset. It is intended to be used with core images to indicate the real memory address. The syntax for *label* is identical to that for *offset*.

SEE ALSO

adb(1)

BUGS

A file name argument can't start with "+". A hexadecimal offset can't be a block count. Only one file name argument can be given.

NAME

proctype – Tell what type of SPU or CONVEX processor is installed

SYNOPSIS

proctype [-c]

DESCRIPTION

Proctype prints nothing. Instead, it sets its SPU OS exit status to indicate the type of SPU or CONVEX processor class. If no options, are specified, **proctype** exits with a 1 for a C1 SPU or a 2 for a C3200/C3400 SPU. If the **–c** option is given, **proctype** exits with a status which indicates one of the following CONVEX machine types:

0	C1 series XP
1	C1 series XL (compact model)
2	C3200 series (2 head)
3	C3200 series (1 head compact model)
4	C3200 series (4 head)
5	C201/C202 series (1 or 2 head)
6	C3232i (up to 3 heads with extra I/O)
7	C3400 (up to 4 heads)
8	C3400 (1 or 2 heads)
9	C3432 (up to 3 heads with extra I/O)

NAME

ps – process status

SYNOPSIS

ps [**aklx**] [namelist]

DESCRIPTION

Ps prints certain indicia about active processes. The **a** option asks for information about all processes with terminals (ordinarily only one's own processes are displayed); **x** asks even about processes with no terminal; **l** asks for a long listing. The short listing contains the process ID, tty letter, the cumulative execution time of the process and an approximation to the command line.

The long listing is columnar and contains:

F Flags associated with the process. **01**: in core; **02**: system process; **04**: locked in core (e.g. for physical I/O); **10**: being swapped; **20**: being traced by another process.

S The state of the process. **0**: nonexistent; **S**: sleeping; **W**: waiting; **R**: running; **I**: intermediate; **Z**: terminated; **T**: stopped.

UID The user ID of the process owner.

PID The process ID of the process; as in certain cults, it is possible to kill a process if you know its true name.

PPID The process ID of the parent process.

PGRP The process group ID of the process.

CPU Processor utilization for scheduling.

PRI The priority of the process; high numbers mean low priority.

NICE Used in priority computation.

ADDR The core address of the process if resident, otherwise the disk address.

SZ The size in blocks of the core image of the process.

WCHAN
The event for which the process is waiting or sleeping; if blank, the process is running.

TTY The controlling tty for the process.

TIME The cumulative execution time for the process.

The command and its arguments.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked <defunct>. **Ps** makes an educated guess as to the file name and arguments given when the process was created by examining core memory or the swap area. The method is inherently somewhat unreliable and in any event a process is entitled to destroy this information, so the names cannot be counted on too much.

If the **k** option is specified, the file */usr/sys/core* is used in place of */dev/mem*. This is used for postmortem system debugging. If a second argument is given, it is taken to be the file containing the system's namelist.

FILES

/unix	system namelist
/dev/mem	core memory
/usr/sys/core	alternate core file
/dev	searched to find swap device and tty names

SEE ALSO

kill(1)

BUGS

Things can change while **ps** is running; the picture it gives is only a close approximation to reality.

Some data printed for defunct processes is irrelevant.

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

SEE ALSO

cd(1)

NAME

reset – reset the teletype bits to a sensible state

SYNOPSIS

reset console
reset tty

DESCRIPTION

Reset sets the terminal to cooked mode, turns off *cbreak* and *raw* modes, turns on *nl*, and restores undefined special characters to their default values. **Reset console** affects both the local console and the modem port, and should be used in most cases. **Reset tty** is useful in multi-user mode, and affects only the user's terminal.

Reset is most useful after a program dies leaving a terminal in a funny state; you have to type "<LF>reset console<LF>" to get it to work, as **<CR>** often doesn't work; often none of this will echo.

SEE ALSO

stty(1)

BUGS

Doesn't set tabs properly; it can't intuit personal choices for interrupt and line kill characters, so it leaves these set to the local system standards.

NAME

rm, rmdir – remove (unlink) files

SYNOPSIS

rm [**–fri**] file ...

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with 'y' the file is deleted, otherwise the file remains. No questions are asked when the **–f** (force) option is given.

If a designated file is a directory, an error comment is printed unless the optional argument **–r** has been used. In that case, **rm** recursively deletes the entire contents of the specified directory, and the directory itself.

If the **–i** (interactive) option is in effect, **rm** asks whether to delete each file, and, under **–r**, whether to examine each directory.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

unlink(2)

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file '..' merely to avoid the antisocial consequences of inadvertently doing something like 'rm –r .*'.

NAME

sh, for, case, if, while, :, ., break, continue, cd, eval, exec, exit, export, login, newgrp, read, readonly, set, shift, times, trap, umask, wait – command language

SYNOPSIS

sh [**–ceiknrstuvx**] [arg] ...

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. See **invocation** for the meaning of arguments to the shell.

Commands.

A *simple-command* is a sequence of non blank *words* separated by blanks (a blank is a **tab** or a **space**). The first word specifies the name of the command to be executed. Except as specified below the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see **exec(2)**). The *value* of a simple-command is its exit status if it terminates normally or 200+*status* if it terminates abnormally (see **signal(2)** for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by **|**. The standard output of each command but the last is connected by a **pipe(2)** to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

A *list* is a sequence of one or more *pipelines* separated by **;**, **&**, **&&** or **||** and optionally terminated by **;** or **&**. **;** and **&** have equal precedence which is lower than that of **&&** and **||**, **&&** and **||** also have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding *pipeline* to be executed without waiting for it to finish. The symbol **&&** (**||**) causes the *list* following to be executed only if the preceding *pipeline* returns a zero (non zero) value. Newlines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a simple-command or one of the following. The value returned by a command is that of the last simple-command executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**
: Each time a **for** command is executed *name* is set to the next word in the **for** word list If **in** *word* ... is omitted then **in "$@"** is assumed. Execution ends when there are no more words in the list.

case *word* **in** [*pattern* [**|** *pattern*] ...) *list* **;;**] ... **esac**
: A **case** command executes the *list* associated with the first pattern that matches *word*. The form of the patterns is the same as that used for file name generation.

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
: The *list* following **if** is executed and if it returns zero the *list* following **then** is executed. Otherwise, the *list* following **elif** is executed and if its value is zero the *list* following **then** is executed. Failing that the **else** *list* is executed.

while *list* [**do** *list*] **done**
: A **while** command repeatedly executes the **while** *list* and if its value is zero executes the **do** *list;* otherwise the loop terminates. The value returned by a **while** command is that of the last executed command in the **do** *list*. **until** may be used in place of **while** to negate the loop termination test.

(*list* **)** Execute *list* in a subshell.

{ *list* **}** *list* is simply executed.

The following words are only recognized as the first word of a command and when not quoted.

if then else elif fi case in esac for while until do done { }

Command substitution.
The standard output from a command enclosed in a pair of grave accents (` `) may be used as part or all of a word; trailing newlines are removed.

Parameter substitution.
The character $ is used to introduce substitutable parameters. Positional parameters may be assigned values by **set**. Variables may be set by writing

name=*value* [*name*=*value*] ...

${*parameter***}**
A *parameter* is a sequence of letters, digits or underscores (a *name*), a digit, or any of the characters * @ # ? - $!. The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is a digit then it is a positional parameter. If *parameter* is * or @ then all the positional parameters, starting with $1, are substituted separated by spaces. $0 is set from argument zero when the shell is invoked.

${*parameter***-***word***}**
If *parameter* is set then substitute its value; otherwise substitute *word*.

${*parameter***=***word***}**
If *parameter* is not set then set it to *word;* the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${*parameter***?***word***}**
If *parameter* is set then substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted then a standard message is printed.

${*parameter***+***word***}**
If *parameter* is set then substitute *word;* otherwise substitute nothing.

In the above *word* is not evaluated unless it is to be used as the substituted string. (So that, for example, echo ${d-`pwd`} will only execute *pwd* if *d* is unset.)

The following *parameters* are automatically set by the shell.

#	The number of positional parameters in decimal.
-	Options supplied to the shell on invocation or by **set**.
?	The value returned by the last executed command in decimal.
$	The process number of this shell.
!	The process number of the last background command invoked.

The following *parameters* are used but not set by the shell.

HOME	The default argument (home directory) for the **cd** command.
PATH	The search path for commands (see **execution**).
MAIL	If this variable is set to the name of a mail file then the shell informs the user of the arrival of mail in the specified file.
PS1	Primary prompt string, by default '$ '.
PS2	Secondary prompt string, by default '> '.
IFS	Internal field separators, normally **space**, **tab**, and **newline**.

Blank interpretation.
After parameter and command substitution, any results of substitution are scanned for internal field separator characters (those found in $IFS) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

File name generation.
Following substitution, each command word is scanned for the characters *, ? and [. If one of these characters appears then the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern then the word is left unchanged. The character . at the start of a file name or immediately following a /, and the character /, must be matched explicitly.

* Matches any string, including the null string.
? Matches any single character.
[...] Matches any one of the characters enclosed. A pair of characters separated by – matches any character lexically between the pair.

Quoting.
The following characters have a special meaning to the shell and cause termination of a word unless quoted.

; & ( ) | < > newline space tab

A character may be *quoted* by preceding it with a \. **\newline** is ignored. All characters enclosed between a pair of quote marks (' '), except a single quote, are quoted. Inside double quotes (" ") parameter and command substitution occurs and \ quotes the characters \ ` " and $.

"$*" is equivalent to "$1 $2 ..." whereas
"$@" is equivalent to "$1" "$2" ... .

Prompting.
When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a newline is typed and further input is needed to complete a command then the secondary prompt ($PS2) is issued.

Input output.
Before a command is executed its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are not passed on to the invoked command. Substitution occurs before *word* or *digit* is used.

< *word* Use file *word* as standard input (file descriptor 0).

> *word* Use file *word* as standard output (file descriptor 1). If the file does not exist then it is created; otherwise it is truncated to zero length.

>> *word*
Use file *word* as standard output. If the file exists then output is appended (by seeking to the end); otherwise the file is created.

<< *word*
The shell input is read up to a line the same as *word*, or end of file. The resulting document becomes the standard input. If any character of *word* is quoted then no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, **\newline** is ignored, and \ is used to quote the characters \ $ ` and the first character of *word*.

<& *digit*
The standard input is duplicated from file descriptor *digit;* see **dup(2)**. Similarly for the standard output using >.

<&– The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit then the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example,

... 2>&1

creates file descriptor 2 to be a duplicate of file descriptor 1.

If a command is followed by & then the default standard input for the command is the empty file (/dev/null). Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input output specifications.

Environment.
The environment is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list; see **exec(2)** and **environ(5)**. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a *parameter* for each name found, giving it the corresponding value. Executed commands inherit the same environment. If the user modifies the values of these *parameters* or creates new ones, none of these affects the environment unless the **export** command is used to bind the shell's *parameter* to the environment. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to *parameters*. Thus these two lines are equivalent

TERM=450 cmd args
(export TERM; TERM=450; cmd args)

If the **–k** flag is set, *all* keyword arguments are placed in the environment, even if the occur after the command name. The following prints 'a=b c' and 'c':
echo a=b c
set –k
echo a=b c

Signals.
The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the shell from its parent. (But see also **trap.**)

Execution.
Each time a command is executed the above substitutions are carried out. Except for the 'special commands' listed below a new process is created and an attempt is made to execute the command via an **exec(2)**.

The shell parameter $PATH defines the search path for the directory containing the command. Each alternative directory name is separated by a colon (:). The default path is **:/bin:/usr/bin**. If the command name contains a / then the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an *b.out* file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a subshell.

Special commands.
The following commands are executed in the shell process and except where specified no input output redirection is permitted for such commands.

: No effect; the command does nothing.

. *file* Read and execute commands from *file* and return. The search path $PATH is used to find the directory containing *file*.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified then break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified then resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter $HOME is the default *arg*.

eval [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input output arguments may appear and if no other arguments are given cause the shell input output to be modified.

exit [*n*]
Causes a non interactive shell to exit with the exit status specified by *n*. If *n* is omitted then the exit status is that of the last command executed. (An end of file will also exit from the shell.)

export [*name* ...]
The given names are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given then a list of exportable names is printed.

login [*arg* ...]
Equivalent to 'exec login arg ...'.

newgrp [*arg* ...]
Equivalent to 'exec newgrp arg ...'.

read *name* ...
One line is read from the standard input; successive words of the input are assigned to the variables *name* in order, with leftover words to the last variable. The return code is 0 unless the end-of-file is encountered.

readonly [*name* ...]
The given names are marked readonly and the values of the these names may not be changed by subsequent assignment. If no arguments are given then a list of all readonly names is printed.

set [**–eknptuvx** [*arg* ...]]

- **–e** If non interactive then exit immediately if a command fails.
- **–k** All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- **–n** Read commands but do not execute them.
- **–t** Exit after reading and executing one command.
- **–u** Treat unset variables as an error when substituting.
- **–v** Print shell input lines as they are read.
- **–x** Print commands and their arguments as they are executed.
- **–** Turn off the **–x** and **–v** options.

These flags can also be used upon invocation of the shell. The current set of flags may be found in $–.

Remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, etc. If no arguments are given then the values of all names are printed.

shift The positional parameters from **$2**... are renamed **$1**...

times Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...
Arg is a command to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. If *arg* is absent then all trap(s) *n* are reset to their original values. If *arg* is the null string then this signal is ignored by the shell and by invoked commands. If *n* is 0 then the command *arg* is executed on exit from the shell, otherwise upon receipt of signal *n* as numbered in **signal(2)**. *Trap* with no arguments prints a list of commands associated with each signal number.

umask [*nnn*]
The user file creation mask is set to the octal value *nnn* (see **umask(2)**). If *nnn* is omitted, the current value of the mask is printed.

wait [*n*]
Wait for the specified process and report its termination status. If *n* is not given then all currently active child processes are waited for. The return code from this command is that of the process waited for.

Invocation.
If the first character of argument zero is –, commands are read from **$HOME/.profile**, if such a file exists. Commands are then read as described below. The following flags are interpreted by the shell when it is invoked.

–c *string* If the **–c** flag is present then commands are read from *string*.

–s If the **–s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

–i If the **–i** flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case the terminate signal SIGTERM (see **signal(2)**) is ignored (so that 'kill 0' does not kill an interactive shell) and the interrupt signal SIGINT is caught and ignored (so that **wait** is interruptable). In all cases SIGQUIT is ignored by the shell.

The remaining flags and arguments are described under the **set** command.

FILES
$HOME/.profile
*/tmp/sh**
/dev/null

SEE ALSO
test(1)
exec(2)

DIAGNOSTICS
Errors detected by the shell, such as syntax errors cause the shell to return a non zero exit status. If the shell is being used non interactively then execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also **exit**).

BUGS
If << is used to provide standard input to an asynchronous process invoked by &, the shell gets mixed up about naming the input document. A garbage file /tmp/sh* is created, and the shell complains about not being able to find the file by another name.

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time as in:

(sleep 105; *command*)&

or to execute a command every so often, as in:

```
while true
do
        command
        sleep 37
done
```

SEE ALSO

alarm(2)
sleep(3)

BUGS

Time must be less than 65536 seconds.

NAME

sort – sort or merge files

SYNOPSIS

sort [**–mubdfinrtx**] [*+pos1* [*–pos2*]] ... [**–o** name] [**–T** directory] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '–' means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b.**

r Reverse the sense of comparisons.

t*x* 'Tab character' separating fields is *x*.

The notation *+pos1 –pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr,** where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first nonblank in the field; **b** is attached independently to *pos2*. A missing *.n* means .0; a missing *–pos2* means the end of the line. Under the **–t***x* option, fields are strings separated by *x*; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort. The input file is *not* sorted, but rather a message indicating the first item that causes disorder is printed. Therefore, it does not make sense to use **–o** in combination with this option.

m Merge only; the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. The **–c** option should not be used along with the **–o** option.

T The next argument is the name of a directory in which temporary files should be made.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

EXAMPLES

Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

```
sort -u +0f +0 list
```

Print the password file (*passwd*(5)) sorted by user id number (the 3rd colon-separated field).

```
sort -t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month day) entries. The options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
sort -um +0 -1 dates
```

FILES

/usr/tmp/stm*, /tmp/* first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **-c**.

BUGS

Very long lines are silently truncated.

NAME

stty – set terminal options

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output terminal. With no argument, it reports the current settings of the options. The option strings are selected from the following set:

even allow even parity
–even disallow even parity
odd allow odd parity
–odd disallow odd parity
raw raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
–raw negate raw mode
cooked same as '–raw'
cbreak make each character available to **read(2)** as received; no erase and kill
–cbreak
make characters available to *read* only when newline is received
–nl allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl accept only new-line to end lines
echo echo back every character typed
–echo do not echo characters
lcase map upper case to lower case
–lcase do not map case
–tabs replace tabs by spaces when printing
tabs preserve tabs
ek reset erase and kill characters back to normal # and @
erase *c* set erase character to *c*. *C* can be of the form *'^X'* which is interpreted as a 'control X'.
kill *c* set kill character to *c*. *'^X'* works here also.
cr0 cr1 cr2 cr3
select style of delay for carriage return (see **ioctl(2)**)
nl0 nl1 nl2 nl3
select style of delay for linefeed
tab0 tab1 tab2 tab3
select style of delay for tab
ff0 ff1 select style of delay for form feed
bs0 bs1 select style of delay for backspace
dec set all modes suitable for Digital Equipment Corp. VT100 terminals
hup hang up dataphone on last close.
–hup do not hang up dataphone on last close.
0 hang up phone line immediately
50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb
Set terminal baud rate to the number given, if possible.

SEE ALSO

ioctl(2)

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [± number[**lbcr**]] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance *+number* from the beginning, or *-number* from the end of the input. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines. Option **r** will display the lines in reverse order.

SEE ALSO

dd(1)

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tar – tape archiver

SYNOPSIS

tar [key] [name ...]

DESCRIPTION

Tar saves and restores files on tape. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r The named files are written on the end of the tape. The **c** function implies this.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, the entire content of the tape is extracted. Note that if multiple entries specifying the same file are on the tape, the last one overwrites all earlier.

t The names of the specified files are listed each time they occur on the tape. If no file argument is given, all of the names on the tape are listed.

u The named files are added to the tape if either they are not already there or have been modified since last put on the tape.

c Create a new tape; writing begins on the beginning of the tape instead of after the last file. This command implies **r.**

The following characters may be used in addition to the letter which selects the function desired.

v Normally **tar** does its work silently. The **v** (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.

w causes **tar** to print the action to be taken followed by file name, then wait for user confirmation. If a word beginning with 'y' is given, the action is performed. Any other input means don't do it.

f causes **tar** to use the next argument as the name of the archive instead of */dev/rmt0.* If the name of the file is '–', tar writes to standard output or reads from standard input, whichever is appropriate. Thus, **tar** can be used as the head or tail of a filter chain **Tar** can also be used to move hierarchies with the command:

cd *fromdir*; **tar cf - . | (cd** *todir*; **tar xf -)**

b causes **tar** to use the next argument as the blocking factor for tape records. The default and the maximum is 238. This option should only be used with raw magnetic tape archives (See **f** above). The block size is determined automatically when reading tapes (key letters 'x' and 't').

l tells **tar** to complain if it cannot resolve all of the links to the files dumped. If this is not specified, no error messages are printed.

m tells **tar** to not restore the modification times. The mod time will be the time of extraction.

FILES

/dev/rct0e
*/tmp/tar**

SEE ALSO

ct(4)
dk(4)

DIAGNOSTICS

Complains about bad key characters and tape read/write errors.
Complains if enough memory is not available to hold the link tables.

BUGS

The **u** and **r** options are not supported at this time.
There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow.
The **b** option should not be used with archives that are going to be updated. If the archive is on a disk file the **b** option should not be used at all, as updating an archive stored in this manner can destroy it.
The current limit on file name length is 100 characters.

NAME

tee – pipe fitting

SYNOPSIS

tee [**–i**] [**–a**] [file] ...

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the *files*. Option **–i** ignores interrupts; option **–a** causes the output to be appended to the *files* rather than overwriting them.

NAME

test – condition command

SYNOPSIS

test expr

DESCRIPTION

Test evaluates the expression *expr*, and if its value is true then returns zero exit status; otherwise, a non zero exit status is returned. **Test** returns a non zero exit if there are no arguments.

The following primitives are used to construct *expr*.

–r file true if the file exists and is readable.

–w file true if the file exists and is writable.

–f file true if the file exists and is not a directory.

–d file true if the file exists and is a directory.

–s file true if the file exists and has a size greater than zero.

–t [fildes]
true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.

–z s1 true if the length of string *s1* is zero.

–n s1 true if the length of the string *s1* is nonzero.

s1 **=** s2 true if the strings *s1* and *s2* are equal.

s1 **!=** s2 true if the strings *s1* and *s2* are not equal.

s1 true if *s1* is not the null string.

n1 **–eq** n2
true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons **–ne**, **–gt**, **–ge**, **–lt**, or **–le** may be used in place of **–eq**.

These primaries may be combined with the following operators:

! unary negation operator

–a binary *and* operator

–o binary *or* operator

(expr)
parentheses for grouping.

–a has higher precedence than **–o.** Notice that all the operators and flags are separate arguments to **test.** Notice also that parentheses are meaningful to the Shell and must be escaped.

SEE ALSO

sh(1)
find(1)

NAME

time – time a command

SYNOPSIS

time command

DESCRIPTION

The given command is executed; after it is complete, **time** prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The times are printed on the diagnostic output stream.

BUGS

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

true, false – provide truth values

SYNOPSIS

true

false

DESCRIPTION

True and **false** are usually used in a Bourne shell script. They test for the appropriate status "true" or "false" before running (or failing to run) a list of commands.

EXAMPLE

```
while true
do
    command list
done
```

SEE ALSO

sh(1)

DIAGNOSTICS

True has exit status zero. **False** has non-zero exit status.

NAME

uptime – UNIX uptime and version print routine

SYNOPSIS

uptime

DESCRIPTION

Uptime prints out the amount of time passed since SP2 UNIX was booted and the revision string that identifies the UNIX release.

FILES

/unix – for addresses of uptime variable and release string address
/dev/kmem – for actual UNIX memory values.

NAME

xed – text editor

SYNOPSIS

xed [–] [name]

DESCRIPTION

Xed is a modification of the standard text editor.

If a *name* argument is given, **xed** simulates an *e* command (see below) on the named file; that is to say, the file is read into **xed**'s buffer so that it can be edited. The optional ' – ' suppresses the printing of character counts by *e*, *r*, and *w* commands.

Xed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

Commands to **xed** have a simple and regular structure: zero or more *addresses* followed by a single character *command*, possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Every command which requires addresses has default addresses, so that the addresses can often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While **xed** is accepting text, it is said to be in *input mode*. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

Xed supports a limited form of *regular expression* notation. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. The regular expressions allowed by **xed** are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression and matches that character.

2. A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.

3. A currency symbol '$' at the end of a regular expression matches the null character at the end of a line.

4. A period '.' matches any character except a new-line character.

5. A regular expression followed by an asterisk '*' matches any number of adjacent occurrences (including zero) of the regular expression it follows.

6. A string of characters enclosed in square brackets '[ ]' matches any character in the string but no others. If, however, the first character of the string is a circumflex '^' the regular expression matches any character except new-line and the characters in the string.

7. The concatenation of regular expressions is a regular expression which matches the concatenation of the strings matched by the components of the regular expression.

8. A regular expression enclosed between the sequences '\(' and '\)'is identical to the unadorned expression; the construction has side effects discussed under the *s* command.

9. The null regular expression standing alone is equivalent to the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by '\'. This also applies to the character bounding the regular expression (often '/') and to '\' itself.

To understand addressing in **xed** it is necessary to know that at any time there is a *current line.* Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character '.' addresses the current line.

2. The character '$' addresses the last line of the buffer.

3. A decimal number *n* addresses the *n*-th line of the buffer.

4. ''*x*' addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.

5. A regular expression enclosed in slashes '/' addresses the first line found by searching toward the end of the buffer and stopping at the first line containing a string matching the regular expression. If necessary the search wraps around to the beginning of the buffer.

6. A regular expression enclosed in queries '?' addresses the first line found by searching toward the beginning of the buffer and stopping at the first line containing a string matching the regular expression. If necessary the search wraps around to the end of the buffer.

7. An address followed by a plus sign '+' or a minus sign '' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.

8. If an address begins with '+' or '–' the addition or subtraction is taken with respect to the current line; e.g. '–5' is understood to mean '.–5'.

9. If an address ends with '+' or '–', then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address '–' refers to the line before the current line. Moreover, trailing '+' and '–' characters have cumulative effect, so '—' refers to the current line less 2.

10. To maintain compatibility with earlier version of the editor, the character '^' in addresses is entirely equivalent to '–'.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma (,). They may also be separated by a semicolon (;). In this case the current line (.) is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/', '?'). The second address of any two-address sequence must

correspond to a line following the line corresponding to the first address.

In the following list of **xed** commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, any command may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed respectively in the way discussed below.

(.)a
<text>
.

The append command reads the given text (starting on the next line) and appends it after the addressed line. '.' is left on the last line input, if there were any, otherwise at the addressed line. Address '0' is legal for this command; text is placed at the beginning of the buffer.

b

The buffer command prints two numbers: the first one is the number of line changes between buffer updates. The second number is the number of changes since the last update. This command causes the buffer to be updated immediately. (If the system crashes when the buffer is current, the edit can be restored via the 'res' (I) command.)

(.,.)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. '. ' is left at the last line input; if there were none, it is left at the first line not deleted.

(.,.) d

The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. '. ' is set to the last line of the buffer. The number of characters read is typed. 'filename' is remembered for possible use as a default file name in a subsequent *r* or *w* command. If there have been changes since the last 'w' command, **xed** will protest. If the 'e' command is reissued, **xed** will comply with the request.

f filename

The filename command prints the currently remembered file name. If 'filename' is given, the currently remembered file name is changed to 'filename'.

(1,$)g/regular expression/command list

In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with '. ' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with '\'. *A*, *i*, and *c* commands and associated input are permitted; the '. ' terminating input mode may be omitted if it would be on the last line of the command list. The (global) commands, *g*, and *v*, are not permitted in the command list.

(.)**i**
<text> .

This command inserts the given text before the addressed line. '. ' is left at the last line input; if there were none, at the addressed line. This command differs from the *a* command only in the placement of the text.

(.)**k***x*

The mark command marks the addressed line with name *x*, which must be a lower-case letter. The address form '' *x* ' then addresses this line.

(.,.)**l**

The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in octal, and long lines are folded. An *l* command may follow any other on the same line.

(.,.)**m***a*

The move command repositions the addressed lines after the line addressed by *a*. The last of the moved lines becomes the current line.

(.,.)**o**

The open command allows intraline editing. Once the line is opened, several single character (no return is required) commands are used. Most of these commands can be preceded by an optional repeat count (denoted below as "n", usually defaulted to 1). The commands are:

n<space> -- move to the right n characters
<return> -- copy down rest of line and exit open mode
nd -- delete n characters
nsx -- search for nth occurrence of the character x
nkx -- kill characters until the nth occurrence of the character x
i...<escape> -- insert the text "..." before the current character
ncx -- change the current n characters to x (must type n characters)
nr...<escape> -- replace n characters with "..." until escape (any number chars)

(.,.)**p**

The print command prints the addressed lines. '. ' is left at the last line printed. The *p* command may be placed on the same line after any command.

q

The quit command causes **xed** to exit. No automatic write of a file is done. However, **xed** will protest if the buffer has been modified since the last 'w' command. A second 'q' will force **xed** to exit.

($)**r** filename

The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The remembered file name is not changed unless 'filename' is the very first file name mentioned. Address '0' is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. '. ' is left at the last line read in from the file.

(.,.)**s**/regular expression/replacement/ or,
(.,.)**s**/regular expression/replacement/**g**

The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator 'g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of '/' to delimit the regular expression and the replacement. '. ' is left at the last line substituted.

An ampersand (&) appearing in the replacement is replaced by the string matching the regular expression. The special meaning of '&' in this context may be suppressed by preceding it by '\'. As a more general feature, the characters '\n ', where *n* is a digit, are replaced by the text matched by the *n* -th regular subexpression enclosed between '\(' and '\)'. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

Lines may be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by '\'.

(. , .) **t** *a*
This command acts just like the *m* command, except that a copy of the addressed lines is placed after address *a* (which may be 0). '. ' is left on the last line of the copy.

(1,$) **v**/regular expression/command list
This command is the same as the global command except that the command list is executed with '. ' initially set to every line *except* those matching the regular expression.

(1,$) **w** filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writeable by everyone). The remembered file name is *not* changed unless 'filename' is the very first file name mentioned. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). '. ' is unchanged. If the command is successful, the number of characters written is typed.

(.) **z** (. | + | - | ,) (n)
Prints a predetermined number of lines (initially 23) anchored at . . If a . is postfixed to 'z', the anchored line will be printed surrounded by context lines both above and below.

($) =
The line number of the addressed line is typed. '. ' is unchanged by this command.

! UNIX command
The remainder of the line after the '!' is sent to UNIX to be interpreted as a command. '. ' is unchanged.

(. +1) <newline>
An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1p'; it is useful for stepping through text.

If an interrupt signal (ASCII DEL) is sent, **xed** prints a '?' and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name, and 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 1 word.

FILES

/tmp/#, temporary; '#' is the process number (in octal).

DIAGNOSTICS

'?' for errors in commands; 'TMP' for temporary file overflow.

BUGS

The *s* command causes all marks to be lost on lines changed.

File formats

This section contains SPU OS file formats.

NAME

fstab – static information about the filesystems

SYNOPSIS

#include <fstab.h>

DESCRIPTION

The file */etc/fstab* contains descriptive information about the various file systems. */etc/fstab* is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. The order of records in */etc/fstab* is important because *fsck, mount,* and *umount* sequentially iterate through */etc/fstab* doing their thing.

The special file name is the **block** special file name, and not the character special file name. If a program needs the character special file name, the program must create it by appending a "r" after the last "/" in the special file name.

If *fs_type* is "rw" or "ro" then the file system whose name is given in the *fs_file* field is normally mounted read-write or read-only on the specified special file. If *fs_type* is "rq", then the file system is normally mounted read-write with disk quotas enabled. The *fs_freq* field is used for these file systems by the **dump(8)** command to determine which file systems need to be dumped. The *fs_passno* field is used by the **fsck(8)** program to determine the order in which file system checks are done at reboot time. The root file system should be specified with a *fs_passno* of 1, and other file systems should have larger numbers. File systems within a drive should have distinct numbers, but file systems on different drives can be checked on the same pass to utilize parallelism available in the hardware.

If *fs_type* is "sw" then the special file is made available as a piece of swap space by the **swapon(8)** command at the end of the system reboot procedure. The fields other than *fs_spec* and *fs_type* are not used in this case.

If *fs_type* is "rq" then at boot time the file system is automatically processed by the **quotacheck(8)** command and disk quotas are then enabled with **quotaon(8)**. File system quotas are maintained in a file "quotas", which is located at the root of the associated file system.

If *fs_type* is specified as "xx" the entry is ignored. This is useful to show disk partitions which are currently not used.

```
#define FSTAB_RW    "rw"    /* read-write device */
#define FSTAB_RO    "ro"    /* read-only device */
#define FSTAB_RQ    "rq"    /* read-write with quotas */
#define FSTAB_SW    "sw"    /* swap device */
#define FSTAB_XX    "xx"    /* ignore totally */

struct fstab {
        char    *fs_spec;   /* block special device name */
        char    *fs_file;   /* file system path prefix */
        char    *fs_type;   /* rw,ro,sw or xx */
        int     fs_freq;    /* dump frequency, in days */
        int     fs_passno;  /* pass number on parallel dump */
};
```

The proper way to read records from */etc/fstab* is to use the routines *getfsent(), getfsspec(), getfstype(),* and *getfsfile().*

FILES

/etc/fstab

SEE ALSO

getfsent(3X)

NAME

group – group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1)
crypt(3)
passwd(1)
passwd(5)

NAME

mtab – mounted file system table

DESCRIPTION

Mtab resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last '/' is thrown away.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(8)

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user-id
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

FILES

/etc/passwd

SEE ALSO

group(5)

NAME

tar – tape archive file format

DESCRIPTION

Tar (the tape archive command) dumps several files into one in a medium suitable for transportation.

A **tar** tape or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the **tar(1)** command line — default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads.

The header block looks like:

```
#define TBLOCK      512
#define NAMSIZ      100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width w) contains w-2 digits, a space, and a null, except *size* and *mtime*, which do not contain the trailing null. *Name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the top bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped. *Chksum* is a decimal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is ASCII '0' if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '2' if it is a symbolic link. The name linked-to, if any, is in *linkname*, with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually rescanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1)

BUGS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by **vi(1)** and **curses(3X)**. Terminals are described in **termcap** by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in **termcap**.

Entries in **termcap** consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode

ei	str		End insert mode; give ":ei=:" if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give ":im=:" if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't overstrike

up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like **ce** \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept–100, is among the more complex entries in the **termcap** file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
        :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
        :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:nd=\E=:\
        :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in **termcap** are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has "automatic margins" (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as **\^** and ****. If it is necessary to place a : in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines which deal with **termcap** use C strings, and strip the high bits of the output very late so that a **\200** comes out as a **\000** would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in **termcap** and to build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **termcap** file to describe it or bugs in *ex*. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in */etc/termcap*. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor. (This only works on version 7 systems.)

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than ^H (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in **termcap** are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the **termcap** file usually assumes that this is on, i.e. **am**.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with *printf*(3S) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the **%** encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is "cm=6\E&%r%2c%2Y". The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cm=^T%.%.". Terminals which use "%." need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit **\t**, **\n** ^**D** and **\r**, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cm=\E=%+ %+ ".

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using **termcap.** The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no

terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **ug** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit

or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the **ko** capability, for example, ":ko=cl,ll,sf,sb:", which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the mime would be **:ma=^Kj^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x***z*.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since **termlib** routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be canceled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1)
curses(3X)
termcap(3X)
tset(1)

vi(1)
ul(1)
more(1)

AUTHOR

William Joy
Mark Horton added underlining and keypad support

BUGS

Ex allows only 256 characters for string capabilities, and the routines in **termcap(3X)** do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

ttys – terminal initialization data

DESCRIPTION

The **ttys** file is read by the *init* program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to **getty(8)**, which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (*Getty* will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, */dev*.

FILES

/etc/ttys

SEE ALSO

init(8)
getty(8)
login(1)

NAME

ttytype – data base of terminal types by port

SYNOPSIS

/etc/ttytype

DESCRIPTION

Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to it. There is one line per port, containing the terminal kind (as a name listed in **termcap(5)**), a space, and the name of the tty, minus */dev/*.

This information is read by **tset(1)** and by **login(1)** to initialize the TERM variable at login time.

SEE ALSO

tset(1)
login(1)

BUGS

Some lines are merely known as "dialup" or "plugboard".

System management

This section contains information related to the SPU system operation and maintenance.

NAME

intro – introduction to system maintenance and operation commands

DESCRIPTION

This section contains information related to system operation and maintenance.

NAME

backup, bldboot, restore – backup/restore SPU disk files program

SYNOPSIS

/etc/backup [-r]
/etc/bldboot file1 file2 [-r] [-f]
/etc/restore

DESCRIPTION

Backup is a script which executes the commands necessary to backup the SPU disk. If the *-r* option is specified, only the root partition is backed up.

The tape drive used by **backup** and **restore** will be */dev/rmt1* (the QIC tape drive) if it exists. Otherwise, the Cipher tape drive will be used. Note that on sp2, the Cipher tape drive is not supported. For **bldboot**, the QIC tape drive will be used by default if */dev/rmt1* exists. However, the **-f** switch can be used to force the use of the Cipher tape drive (C-1 only).

Bldboot is the program that is executed to backup the boot tracks and the root image in binary form and, if the *-r* is not specified, the mounted file system(s) in **tar**(1) format. *File1* is the copy of the cartridge tape boot track that is stored on the root file system. *File2* is the copy of the disk/tape format utility used to format and build the disk and tape on the SPU. The **bldboot** program will place the following data on the cartridge tape:

Four copies of the cartridge tape boot track program. These copies are used by the SPU EPROM program. The copies should be writable in about 25 seconds on the Cipher tape drive (C-1 only), or about 3 seconds on the QIC tape drive.

Two copies of the disk/tape format utility (spu2000). The tape boot track program will load copy 0 first and if errors are detected the second copy will be used, if possible. The copies should be writable in about a minute on the Cipher tape drive (C-1 only), or about 5 seconds on the QIC tape drive.

Two copies of the boot/root binary image. The boot tracks and the root file system are copied to the tape twice. The disk/tape format routine will load the copy the user selected. Each copy should be writable in about six minutes on the Cipher tape drive (C-1 only), or about one minute on the QIC tape drive.

The *tar* of the mount file system(s) will take an amount of time directly related to the number of files.

Restore will finish the rebuild procedure. **Restore** is a script that will build the mount file system(s) by executing a **mkfs**(8) and a **mklost+found**(8) on the mount partition(s). The **restore** script will then execute a **tar**(1) command to restore the mount file system(s). The time it takes to restore the file system will be related to how many files are on the tape.

FILES

/bt0.cart	the cartridge tape boot program
/stand/spu2000.cart	the disk/tape format utility
/dev/dk0a	boot tracks and root file system source (Winchester disk)
/dev/dk0d	mount system partition (Winchester disk)
/dev/dk0e	mount system partition 2 (sp2 only)
/dev/rct0b	the partition on the cartridge tape for boot/root backup (Cipher tape, C-1 only)
/dev/rmt1	QIC tape drive
/mnt	the /dev/dk0d file system root node name

/hw the /dev/dk0e file system root node name (sp2 only)
/dev/dk2a boot tracks and root file system source (IOmega disk)
/dev/dk3a mount file system partition (IOmega disk)
/dev/dk2d mount file system partition (IOmega disk)

SEE ALSO

tar(1)
format(8)
backup(5)

NAME

cleanup – clean up SPU disk prior to shipment

SYNOPSIS

/etc/cleanup

DESCRIPTION

/etc/cleanup is used to remove internal directories and files from the SPU disk prior to shipment. This command should be executed as part of shipment preparation in order to prevent internal software from being inadvertantly distributed to the field.

NAME

fasthalt – reboot the SP2, disabling /etc/fsck on the next reboot

SYNOPSIS

/etc/fasthalt

DESCRIPTION

Fasthalt is a shell script that disables the execution of **/etc/fsck** on the next boot, then performs an **/etc/reboot**. The disabling of the filesystem checks is good for one reboot only. Use **/etc/fasthalt** instead of **/etc/reboot** to reboot the SPU without performing filesystem checks.

NAME

format – How to format the SPU disk

DESCRIPTION

Warning: This procedure is for formatting the SPU disk only. This procedure should not be attempted if a backup tape of the SPU does not exist. (See **backup(8)**)

To format the SPU disk for UNIX is a two step process. The need for each process is a function of the state of the SPU disk.

The first process is to format the SPU disk using the spu2000 program. If the disk has a correct format then this process can be skipped. The disk could have a proper format, but is just missing the key data on it required for SPU UNIX to operate. If it has been determine that the disk format is bad then the following steps should be taken.

1. Obtain a cartridge tape that has **backup(8)** data format on it and load it in the SPU tape unit.

2. Boot from this cartridge tape and execute the disk format program in the manner shown below.

```
Convex-1 Front Panel V1.0 / CPU Serial Number 65535 / Switch Register = 34A0
mode-of-operation = diagnostic        boot-device = disk
location-of-bootstrap = default       power-up-reboot = enable
automatic-reboot = enable             spu-selftest = disable
os-flags = 20
(fp)> set boot-device=tape
(fp)> boot

SPU cartridge tape boot (SPU rev C) $Revision: 1.1 $

Reading bad block table 0
Loading copy #0... 21696(32768)+5972(54464)+139944  start: 0x8000

SPU Disk/Tape Diagnostic Utility $Revision: 1.1 $
  (U) for UNIX Root Restore
  (D) for Disk/Tape Format Utility
  (S) for SPU Hardware Utility
  (R) for Reboot SPU
Enter utility to execute - d

SPU Format Disk/Tape Utility
  (D) for Disk (SPU winchester)
  (T) for Tape (SPU cartridge)
  (E) for Exit test
Enter controller type/function - d
        Number of heads ---------- (6)
        Number of cylinders ------ (320)
        Start of write precomp --- (128)
        Step rate ---------------- (2)
        Sector data size --------- (512)
        Sectors per Track/Head --- (18)
        Logical drive number ----- (0)
```

```
    Sector Interleave ------- (3)
Format, Debug or Abort operation [F,D,A]? f
      Run format test? ------- [yn] (n) y
      Run write test? -------- [yn] (n) y
      Run read test? --------- [yn] (n) y
      Run bad block fix? ----- [yn] (n) y
      Run random read test? -- [yn] (n) y
      Run seek test? --------- [yn] (n) y
      LOOP ON TESTS? --------- [yn] (n)
      MAX NUMBER OF ERRORS --- (1)
```

The disk format operation will take about 25 minutes. The build bad block subtest will require user input data in the form of user defined bad blocks (if any). Each subtest will report execution status at the end of the subtest. Any deviation from the example given will cause unpredictable results.

Once the format has been completed or if the disk format was assumed correct the following steps can be attempted.

1. Obtain a cartridge tape that has **backup(8)** data format on it and load it in the SPU tape unit.

2. Boot from this cartridge tape and execute the disk format program in the manner shown below.

```
Convex-1 Front Panel V1.0 / CPU Serial Number 65535 / Switch Register = 34A0
mode-of-operation = diagnostic          boot-device = disk
location-of-bootstrap = default         power-up-reboot = enable
automatic-reboot = enable               spu-selftest = disable
os-flags = 20
(fp)> set boot-device=tape
(fp)> boot

SPU cartridge tape boot (SPU rev C) $Revision: 1.1 $

Reading bad block table 0
Loading copy #0... 21696(32768)+5972(54464)+139944  start: 0x8000

SPU Disk/Tape Diagnostic Utility $Revision: 1.1 $
  (U) for UNIX Root Restore
  (D) for Disk/Tape Format Utility
  (S) for SPU Hardware Utility
  (R) for Reboot SPU
Enter utility to execute - u

SPU UNIX Root Partition Restore
   reading bad block table...0
   reading root date code...0

SPU UNIX root size = 3078 blocks. Backed up Mon Jul 23 12:24:18 1984

 Define the winchester disk to recover
      Number of heads --------- (6)
```

```
Number of cylinders ------ (320)
Start of write precomp --- (128)
Step rate ---------------- (2)
Sector data size --------- (512)
Sectors per Track/Head --- (18)
Logical drive number ----- (0)
Sector Interleave -------- (3)

Recover the root at this time? [yn] (n)y
Recover copy 0 or 1? (01) [0]
```

These steps will take around 3 minutes to complete and after completion the disk will have a complete boot track image and the root partition. The SPU should be rebooted from the disk at this time to complete the disk format.

The following step should be taken to finish the disk rebuild.

```
SPU Disk/Tape Diagnostic Utility $Revision: 1.1 $
  (U) for UNIX Root Restore
  (D) for Disk/Tape Format Utility
  (S) for SPU Hardware Utility
  (R) for Reboot SPU
Enter utility to execute - r

Convex-1 Front Panel V1.0 / CPU Serial Number 65535 / Switch Register = 35A0
mode-of-operation = diagnostic        boot-device = disk
location-of-bootstrap = default       power-up-reboot = enable
automatic-reboot = enable             spu-selftest = disable
os-flags = 20
(fp)> set boot-device=disk
(fp)> boot
```

Once SPU UNIX has booted the **restore(8)** program will need to be executed.

SEE ALSO

backup(8)

BUGS

The disk format/rebuild program spu2000 does not execute under UNIX and because it is a standalone program it does not duplicate UNIX interfaces exactly.

NAME

fsck – file system consistency check and interactive repair

SYNOPSIS

/etc/fsck –p [filesystem ...]
/etc/fsck [**–y**] [**–n**] [**–s**X] [**–SX**] [**–t** filename] [filesystem] ...

DESCRIPTION

The first form of **fsck** preens a standard set of file systems or the specified file systems. It is normally used in the script */etc/rc* during automatic reboot. In this case **fsck** reads the table */etc/fstab* to determine which file systems to check. It uses the information there to inspect groups of disks in parallel taking maximum advantage of i/o overlap to check the file systems as quickly as possible. Normally, the root file system will be checked on pass 1, other "root" ("a" partition) file systems on pass 2, other small file systems on separate passes (e.g. the "d" file systems on pass 3 and the "e" file systems on pass 4), and finally the large user file systems on the last pass, e.g. pass 5. A pass number of 0 in fstab causes a disk to not be checked; similarly partitions which are not shown as to be mounted "rw" or "ro" are not checked.

The system takes care that only a restricted class of innocuous inconsistencies can happen unless hardware or software failures intervene. These are limited to the following:

Unreferenced inodes

Link counts in inodes too large

Missing blocks in the free list

Blocks in the free list also in files

Counts in the super-block wrong

These are the only inconsistencies which **fsck** with the **–p** option will correct; if it encounters other inconsistencies, it exits with an abnormal return status and an automatic reboot will then fail. For each corrected inconsistency one or more lines will be printed identifying the file system on which the correction will take place, and the nature of the correction. After successfully correcting a file system, **fsck** will print the number of files on that file system and the number of used and free blocks.

Without the **–p** option, **fsck** audits and interactively repairs inconsistent conditions for file systems. If the file system is inconsistent, the operator is prompted for concurrence before each correction is attempted. It should be noted that a number of the corrective actions which are not fixable under the **–p** option will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission **fsck** will default to a **–n** action.

Fsck has more consistency checks than its predecessors *check, dcheck, fcheck,* and *icheck* combined.

The following flags are interpreted by **fsck.**

–y Assume a yes response to all questions asked by **fsck**; this should be used with great caution as this is a free license to continue after essentially unlimited trouble has been encountered.

–n Assume a no response to all questions asked by **fsck**; do not open the file system for writing.

–sX Ignore the actual free list and (unconditionally) reconstruct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done; if this is not possible, care should be taken that the system is quiescent and that it is rebooted immediately afterwards. This precaution is necessary so that the old, bad, in-

core copy of the superblock will not continue to be used, or written on the file system.

The **–s***X* option allows for creating an optimal free-list organization. The following forms of *X* are supported for the following devices:

–s3 (RP03)
–s4 (RP04, RP05, RP06)
–sBlocks-per-cylinder:Blocks-to-skip (for anything else)

If *X* is not given, the values used when the filesystem was created are used. If these values were not specified, then the value *400:9* is used.

–S*X* Conditionally reconstruct the free list. This option is like **–s***X* (above) except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using **–S** will force a no response to all questions asked by **fsck.** This option is useful for forcing free list reorganization on uncontaminated file systems.

–t If **fsck** cannot obtain enough memory to keep its tables, it uses a scratch file. If the **–t** option is specified, the file named in the next argument is used as the scratch file, if needed. Without the **–t** flag, **fsck** will prompt the operator for the name of the scratch file. The file chosen should not be on the filesystem being checked, and if it is not a special file or did not already exist, it is removed when **fsck** completes.

If no filesystems are given to **fsck** then a default list of file systems is read from the file **/etc/fstab**.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Directory size not 16-byte aligned.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.
8. Super Block checks:
   More than 65536 inodes.
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the **lost+found** directory. The name assigned is the inode number. The only restrictions are that the directory **lost+found** must preexist in the root of the filesystem being checked and there must be empty slots in which entries can be made. This is accomplished by making **lost+found**, copying a number of files to the directory, and then removing them (before **fsck** is executed).

Checking the raw device is almost always faster.

FILES

/etc/fstab contains default list of file systems to check.

DIAGNOSTICS

The diagnostics produced by **fsck** are intended to be self-explanatory.

SEE ALSO

fstab(5) reboot(8)

BUGS

Inode numbers for . and .. in each directory should be checked for validity.

-g and **-b** options from *check* should be available in **fsck**.

There should be some way to start a **fsck -p** at pass *n*.

NAME

getty – set typewriter mode

SYNOPSIS

/etc/getty [char]

DESCRIPTION

Getty is invoked by **init(8)** immediately after a typewriter is opened. It 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 **ttys(5)** 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.

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.

The following arguments from the *ttys* file are understood.

0 Cycles through 300-1200-150-110 baud. Useful as a default for dialup lines accessed by a variety of terminals.

– Intended for an on-line Teletype model 33, for example an operator's console.

1 Optimized for a 150-baud Teletype model 37.

2 Intended for an on-line 9600-baud terminal, for example the Textronix 4104.

3 Starts at 1200 baud, cycles to 300 and back. Useful with 212 datasets where most terminals run at 1200 speed.

5 Same as '3' but starts at 300.

4 Useful for on-line console DECwriter (LA36).

SEE ALSO

init(8)
ioctl(2)
ttys(5)

NAME

init - process control initialization

SYNOPSIS

/etc/init

DESCRIPTION

Init is invoked as the last step of the boot procedure (see **reboot(8)**). 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. **Init** catches the hangup signal SIGHUP and interprets it to mean that the system should be brought from multi user to single user. Use 'kill -1 1' to send the hangup signal.

FILES

/dev/tty?
/etc/ttys

SEE ALSO

kill(1)
sh(1)
ttys(5)
getty(8)

NAME

installsw – install or create CONVEX software release tapes

SYNOPSIS

/etc/installsw [-r] [-i] [-f in_file] [-d tape_name]

DESCRIPTION

Installsw is used to create and install C-1 and SPU software release tapes. **Installsw** may be run on the C-1 or the SPU.

A release tape contains a header file with the CONVEX copyright notice, and other information describing the software release; a script file to perform software installation; and one or more data files to be installed by the installation script. The installation process involves verification of the header file followed by loading and execution of the installation script file. The script file controls the remainder of the installation process. The creation of an install tape involves specification of text for the header and install script, plus specification of a build script which is executed to control creation of the data portion of the release tape. The build script is not written to the tape.

The default creation/installation device for CONVEX software is */dev/rmt12*. This can be changed by the **device** command shown below, or the **-d** option on the command line. It can be changed to the SPU cartridge tape by using the device name *ct*. When running on the SPU, the creation/installation device is the SPU cartridge tape only.

The **-r**, **-i**, and **-f** options are mutually exclusive.

If the **-r** option is used, the **verify** command (see below) is executed noninteractively to verify the tape header. The default device is used unless specified with **-d** option.

If the **-i** option is used, the **install** command (see below) is automatically executed noninteractively. The default device is used unless specified with **-d** option.

If the **-f** option is used, **installsw** takes all further command input from the file *in_file*. The input must contain valid commands from the list given below. All commands are executed noninteractively. The default device is used unless specified with the **-d** option. Of course, the command file can also respecify the tape device.

If **installsw** is invoked without the **-r**, **-i**, or **-f** commands then it operates in interactive mode. The initial tape device setting is the default device unless changed with the **-d** option on the command line. The user can also specify the tape device at any time with the **device** command.

When in interactive mode, the user is prompted for commands. Command execution may involve considerably more user interaction. This is true even for command files executed from interactive mode (with the **auto** command).

The following commands may be input directly to the interactive prompt, or used in commands files invoked with the **auto** command, or the **-f** option on the command line.

d[evice] *dev_name*

This command specifies the device to be used by **install** or **create**. If the device is not set with this command, the default specified in the opening paragraphs applies, unless changed by the **-d** option on the **installsw** command line. If *ct* is specified (from the C-1), the SPU cartridge tape will be accessed.

Note: **create** in interactive mode also allows the device name to be changed.

Device with no arguments will print out the current *dev_name*.

h[eader] */hd_file/*

s[cript] */scr_file/*

b[uild] */bld_file/*

These commands specify files to be used for header text, install script, and build script when creating a tape. The files are copied to temporary workfiles for later use by the **create** command. The **edit** and **create** commands allow these temporary copies to be edited.

If no file name is given, and **installsw** is in interactive mode, the temporary file is created from keyboard input. Input is terminated with **^D** as the first character in a line. The **e[dit]** command can also be used to create these temporary files.

e[dit] [h] [s] [b]

This command allows the user to edit or create the temporary copies of *hd_file*, *scr_file*, and *bld_file*, respectively. More than one option maybe specified. No options, means edit all files. On the C-1 system, the user's favorite editor, as specified by the EDITOR environment string, is used. If this string is not specified, "vi" is used. On the SPU, "xed" is used.

This command can only be used in interactive mode.

v[erify] [h] [s] [b] [t]

This command displays the contents of the temporary copies of *hd_file*, *scr_file*, and *bld_file*, respectively; or, for the **t** option, the header and script files from the tape at the current *dev_name*. If no option is given, only the header file from the tape is displayed.

When displaying the header file on tape, the copyright notice and system generated date/time stamp will be seen prepended to the user's header file.

c[reate]

Create an **installsw** tape. If in interactive mode, the user is allowed to edit the work copies of *hd_file*, *scr_file*, and *bld_file*. He is also allowed to change *dev_name* for C-1 software. When the user considers all of these data acceptable, **create** writes the temporary copies of the header and script files to the release tape. (The header file is prepended with the standard copyright notice and system date/time stamp). All user interaction is skipped if **installsw** is not in interactive mode. After the header and install script have been written, then the shell script in *bld_file* is executed. The first argument supplied to the script is the path name to which the data is to be written. For magnetic tape on the C-1, or cartridge tape on the SPU, data files can be written to the device directly or via any utility (such as *tar*). For SPU cartridge tape access from the C-1, the *bld_file* script must create one or more temporary files and use the special script *ctar_ct* to transfer the files to the device at the supplied path name. *Ctar_ct* allows users who are not superusers to write to the SPU cartridge tape using *ctar*. (The command line for *ctar_ct* is the same as for *ctar*.)

i[nstall]

This command installs a tape. The device used may be set with the **d** command, or the **-d** option on the **installsw** command line. The header file is checked to see if it is a legal **installsw** tape. If so, the header file is displayed on the standard output. The user sees the header file prepended with the standard copyright notice and the date/time stamp when the release tape was created. If in interactive mode, the user is prompted for

permission to continue. If the response is yes, the script file is read from the tape and executed. The first argument **installsw** supplies to the script is the pathname of the installation device. If using standard magnetic tape from the C-1, or cartridge tape from the SPU, the data files may be read directly by the script. If using cartridge tape from the C-1, the script file should use *ctar_ct* to load the necessary files, and then perform any moving or renaming necessary.

a[uto] *in_file*
This command is the same as specifying the **-f** argument when invoking **installsw**. Further commands are taken from *in_file*, until an end-of-file is reached. **Auto** commands can be nested. *In_file* may also be terminated by **quit** or a period on a line by itself.

When **installsw** is in interactive mode, command files are run with full user interaction. (Command files executed by invoking **installsw** with the **-f** option are run noninteractively.)

q[uit] and **.** ("period")
Terminate the current command file. Terminate **installsw** if at the prompt **->**. Command files are automatically terminated by reaching EOF (**quit** or period in a command file is unnecessary).

In interactive mode **installsw** may also be terminated by responding to the prompt with a **^D** (control-d).

!shell_cmd
Execute the shell command following the !.

? Print a list of the available commands to the standard output.

The following text is a comment. No processing is performed. This is provided to allow commentary inside command files.

USER BUILD AND INSTALL SCRIPTS

User *bld_file* and *scr_file* scripts will be executed by the standard Bourne shell. All facilities for controlling the execution of the shell are described in the shell documents.

Installsw attempts to gather the exit status from script executions. If error status is returned via "exit" commands, it will be reported, and the **installsw** execution will be terminated. Note that some commands (such as *tar*) that may be used in these scripts do not always report proper status on all exceptions. This decreases the utility of full status handling by **installsw**, but we do our best.

DEVICES

When executed on the CONVEX CPU, **installsw** may utilize magnetic tape drives, with */dev/rmt12* as the default drive. It is also possible to access the cartridge tape on the SPU from the C-1. This is done by specifying device name *ct*.

When executed on the SPU, **installsw** uses only the SPU cartridge tape drive.

The following note applies to cartridge tape usage: The header and script files are written using *tar* or *ctar_ct* onto file */dev/rct0b*, from disk files */tmp/install1* and */tmp/install2*, respectively.

This way they can be recovered in another **installsw** session by using the same file names. The install data is written by the *bld_file* script onto file */dev/rct0c*. When writing from the C-1 system onto the SPU cartridge tape, the user supplied script must use the *ctar_ct* utility. The data may be up to approximately 17 Megabytes.

COOKBOOK APPROACH

As a straight forward method of using **installsw** on the SPU, the following procedure is suggested. Four files are required:

foo_in_file	Installsw command input file
foo_header	Tape header file
foo_build	Tape build script
foo_install	Tape install script

The **installsw** command file, *foo_in_file*, should contain the following commands:

```
device  ct
header  foo_header
script  foo_install
build   foo_build
verify  h
create
quit
```

The tape header file, *foo_header*, can contain any ASCII text and should specify useful information about the tape such as the contents of the tape and the tape release date. For example:

Product:	System Diagnostics V2.0
Release date:	Jul 1 1985
Directories:	/mnt/bin, /mnt/test

The tape build script, *foo_header*, should consist of the commands to build the tape:

```
:
: System Diagnostics Build Script
:
tar cv bin test
```

Finally, the tape install script, *foo_install*, should consist of all commands necessary to install the tape on the target system:

```
:
: System Diagnostics Installation Script
:
tar xvf /dev/rct0b /tmp/install1
cd /mnt
rm -rf DIAG_REV bin test
tar xv bin test 2>&1 | tee /tmp/installsw.tar
mv /tmp/install1 DIAG_REV
echo "System Diagnostics installation complete"
rm -f /tmp/install1 /tmp/install2
exit 0
```

Note that since **installsw** checks the status returned by *foo_build* and *foo_install*, it is good practice to include an **exit 0** command at the end of the script. This prevents non-zero exit status from being returned inadvertently as can happen if certain UNIX commands (such as **test**) are the last commands to be executed by the script.

Having established these four files, **installsw** format tapes can be created by executing the command:

/etc/installsw -f foo_in_file

The resultant tape can then be installed on another system via the command:

/etc/installsw -i

FILES

Temporary files are:

*/tmp/Ins_B**
*/tmp/Ins_H**
*/tmp/Ins_S**
*/tmp/Ins_Y**
*/tmp/Ins_Z**
/tmp/install1
/tmp/install2

NAME

mkfs – construct a file system

SYNOPSIS

/etc/mkfs special proto

DESCRIPTION

Mkfs constructs a file system by writing on the special file *special* according to the directions found in the prototype file *proto*. The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program, see **bproc(8)**. The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of i-nodes in the i-list. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters –**bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or – to specify set-user-id mode or not. The third is **g** or – for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see **chmod(1)**.

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, **mkfs** makes the entries . and .. and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token $.

If the prototype file cannot be opened and its name consists of a string of digits, **mkfs** builds a file system with a single empty directory on it. The size of the file system is the value of *proto* interpreted as a decimal number. The number of i-nodes is calculated as a function of the filsystem size. The boot program is left uninitialized.

A sample prototype specification follows:

```
/usr/mdec/uboot
4872 55
d—777 3 1
usr     d—777 3 1
        sh      ——755 3 1 /bin/sh
        ken     d—755 6 1
                $
        b0      b—644 3 1 0 0
        c0      c—644 3 1 0 0
        $
$
```

SEE ALSO

filsys(5)
dir(5)

BUGS

There should be some way to specify links.

NAME

mklost+found – make a lost+found directory for fsck

SYNOPSIS

/etc/mklost+found

DESCRIPTION

A **lost+found** directory is created in the current directory and a number of empty files are created therein and then removed so that there will be empty slots for **fsck(8)**. This command should not normally be needed since **mkfs(8)** automatically creates the **lost+found** directory when a new file system is created.

SEE ALSO

fsck(8)
mkfs(8)

NAME

mknod – build special file

SYNOPSIS

/etc/mknod name [**c**] [**b**] major minor

DESCRIPTION

Mknod makes a special file. The first argument is the *name* of the entry. The second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number).

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file *conf.c*.

SEE ALSO

mknod(2)

NAME

mount, umount – mount and dismount file system

SYNOPSIS

/etc/mount [special name [**–r**]]

/etc/mount –a

/etc/umount special

/etc/umount –a

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special.* The file *name* must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root. The optional argument **–r** indicates that the file system is to be mounted read-only.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

If the **–a** option is present for either **mount** or **umount,** all of the file systems described in */etc/fstab* are attempted to be mounted or unmounted. In this case, *special* and *name* are taken from */etc/fstab*. The *special* file name from */etc/fstab* is the block special name.

These commands maintain a table of mounted devices in */etc/mtab*. If invoked without an argument, **mount** prints the table.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

FILES

/etc/mtab	mount table
/etc/fstab	file system table

SEE ALSO

mount(2)
mtab(5)
fstab(5)

BUGS

Mounting file systems full of garbage will crash the system.
Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

NAME

pstat – print system facts

SYNOPSIS

pstat [**–aixptuf**] [suboptions] [file]

DESCRIPTION

Pstat interprets the contents of certain system tables. If *file* is given, the tables are sought there, otherwise in */dev/mem*. The required namelist is taken from */unix*. Options are

–a Under **–p**, describe all process slots rather than just active ones.

–i Print the inode table with the these headings:

LOC The core location of this table entry.

FLAGS Miscellaneous state variables encoded thus:

- L locked
- U update time **filsys(5)**) must be corrected
- A access time must be corrected
- M file system is mounted here
- W wanted by another process (L flag is on)
- T contains a text file
- C changed time must be corrected

CNT Number of open file table entries for this inode.

DEV Major and minor device number of file system in which this inode resides.

INO I-number within the device.

MODE Mode bits, see **chmod(2)**.

NLK Number of links to this inode.

UID User ID of owner.

SIZ/DEV

Number of bytes in an ordinary file, or major and minor device of special file.

–x Print the text table with these headings:

LOC The core location of this table entry.

FLAGS Miscellaneous state variables encoded thus:

- T **ptrace(2)** in effect
- W text not yet written on swap device
- L loading in progress
- K locked
- w wanted (L flag is on)

DADDR Disk address in swap, measured in multiples of 512 bytes.

CADDR Core address, measured in multiples of 64 bytes.

SIZE Size of text segment, measured in multiples of 64 bytes.

IPTR Core location of corresponding inode.

CNT Number of processes using this text segment.

CCNT Number of processes in core using this text segment.

–p Print process table for active processes with these headings:

LOC The core location of this table entry.

S Run state encoded thus:

- 0 no process
- 1 waiting for some event
- 3 runnable
- 4 being created

5 being terminated
6 stopped under trace

F Miscellaneous state variables, or-ed together:
01 loaded
02 the scheduler process
04 locked
010 swapped out
020 traced
040 used in tracing
0100 locked in by **lock(2)**.

PRI Scheduling priority, see **nice(2)**.

SIGNAL Signals received (signals 1-16 coded in bits 0-15),

UID Real user ID.

TIM Time resident in seconds; times over 127 coded as 127.

CPU Weighted integral of CPU time, for scheduler.

NI Nice level, see **nice(2)**.

PGRP Process number of root of process group (the opener of the controlling terminal).

PID The process ID number.

PPID The process ID of parent process.

ADDR If in core, the physical address of the 'u-area' of the process measured in multiples of 64 bytes. If swapped out, the position in the swap area measured in multiples of 512 bytes.

SIZE Size of process image in multiples of 64 bytes.

WCHAN Wait channel number of a waiting process.

LINK Link pointer in list of runnable processes.

TEXTP If text is pure, pointer to location of text table entry.

CLKT Countdown for **alarm(2)** measured in seconds.

–t Print table for terminals (only DH11 and DL11 handled) with these headings:

RAW Number of characters in raw input queue.

CAN Number of characters in canonicalized input queue.

OUT Number of characters in putput queue.

MODE See **tty(4)**.

ADDR Physical device address.

DEL Number of delimiters (newlines) in canonicalized input queue.

COL Calculated column position of terminal.

STATE Miscellaneous state variables encoded thus:
W waiting for open to complete
O open
S has special (output) start routine
C carrier is on
B busy doing output
A process is awaiting output
X open for exclusive use
H hangup on close

PGRP Process group for which this is controlling terminal.

–u print information about a user process; the next argument is its address as given by **ps(1)**. The process must be in main memory, or the file used can be a core image and the address 0.

–f Print the open file table with these headings:

LOC The core location of this table entry.

FLG Miscellaneous state variables encoded thus:
R open for reading

W open for writing
P pipe

CNT Number of processes that know this open file.

INO The location of the inode table entry for this file.

OFFS The file offset, see **lseek(2)**.

FILES

/unix namelist

/dev/mem default source of tables

SEE ALSO

ps(1)
stat(2)
filsys(5)
K. Thompson, *UNIX Implementation*

NAME

pwrdwn – power down the CONVEX-1 system

SYNOPSIS

pwrdwn

DESCRIPTION

Pwrdwn will verify that a cartridge tape is not loaded in the drive, will send an interrupt signal (see **signal(2)**) to all UNIX processes, will sync the SPU disk, and will position the SPU disk heads to the shipping zone. After all of the above steps succeed the console will print:

pwrdwn: Ready for power down. ^D to abort

At this time the system power can be turned off or the command can be aborted using the "control-D" key.

BUGS

Some processes may not go away if sent an interrupt signal. To be safe the **ps(1)** command should be run before **pwrdwn** to be sure that only the "normal" UNIX processes are running.

NAME

reboot – SPU Unix bootstrapping procedures

SYNOPSIS

/etc/reboot [-n]

DESCRIPTION

SPU Unix is started by placing the executable image of Unix in SPU memory and transferring control to its main entry point. Since SPU Unix is not reenterable, it is necessary to read Unix in from disk or tape each time it is to be bootstrapped.

Rebooting a Running System

SPU Unix can be rebooted by executing */etc/reboot* from the system console. **Reboot** will sync the file systems and make a **reboot(2)** call to cause UNIX to exit to the SPU front panel monitor which is EPROM resident. Processes running will not be given a chance to exit and if **-n** is specified the file system may be corrupted if all files are not closed. Thus, it is suggested that one execute **ps(1)** to determine what processes are active and then **kill(1)** all processes other than the shell, **sh(1).** The option **-n** will cause reboot to not sync the file system before calling **reboot(2)**.

The front panel monitor will display the status of various soft switches as shown below and then prompt for user input.

```
CONVEX Front Panel - Version: 3.01 / CPU Class: 7 / CPU SN 32514
SPU type = SP5                          Processor = 68000
mode-of-operation = normal-os           boot-device = disk
location-of-bootstrap = default         power-up-reboot = enable
automatic-reboot = enable               spu-selftest = disable
test-flags = normal                     remote-port-bps = 1200
SCSI-power-up-delay = 0xA               user-flags = 0x0
(fp)> boot
SPU OS boot
: <CR>
```

For the typical reboot scenario, the user should respond to the front panel monitor prompt, **(fp)>,** by entering the command **boot** . This will load and execute the SPU OS primary boot program which resides in the otherwise unused block zero of the boot device. When executed, the primary boot program will initialize SPU memory management and, if the mode-of-operation is set to diagnostic, prompt with a ":" on the system console for the device/file specification of the SPU OS image to be loaded. A device/file specification has the following form:

device(unit,offset)pathname

where *device* is the type of device to be searched, *unit* is the unit number of the device, *offset* is the block offset of the file system on the device, and *pathname* is the directory pathname of the file to be loaded. *Device* is one of the following:

dk SPU Winchester disk

The default device/file specification is

dk(1,0)unix

and is selected automatically if a carriage return, <CR>, is entered in response to the SPU Unix boot prompt. The primary boot program will find the corresponding file on the given device, load the file into memory, and transfer control to that file.

Once SPU Unix has booted, a complete file system check will automatically be performed. If an error is detected in the root partition, an automatic reboot will be initiated in which case the front panel monitor program will be entered once again. The user should repeat the above procedure by entering the **boot** command. If errors are detected in any mounted partitions, then the user will be instructed to execute **fsck(8)** manually to resolve the problems. Once the file system has been verified, SPU Unix will prompt with **(spu)>**.

Cold starts. The user should be certain that the front panel mode switch is in the "local" or "secure" positions. If the mode switch is in the "remote" position, the system console keyboard will be disabled and no command entry will be possible. On power up, the SPU begins execution of its EPROM resident code. First, the SPU self test is executed if enabled, and then control passes to the front panel monitor program. The front panel monitor will display the status of various soft switches and prompt for a command. A menu of possible commands can be displayed by entering the **help** command. For the normal cold boot of SPU OS the user should insure that the switch settings are as shown above. If not, use the **help** and **set** commands to change the switch values shown above and then execute the **boot** command. The boot process will proceed as described above.

In the event the a copy of the primary boot program is unreadable, an alternate copy can be selected by returning to the front panel monitor by pressing the reset button on the front panel of the system. The front panel monitor **set** command can then be used to specify that an alternate copy of the primary boot program be used. There are three copies of the primary boot program on disk and four copies of the primary boot program on a backup tape generated by the **backup(8)** command. The copies are referred to as default, 1st-copy, 2nd-copy, and 3rd-copy, respectively. The copy used for the primary boot can be selected by setting the *location-of-bootstrap* switch to *default*, *1st-copy*, *2nd-copy*, or *3rd-copy* (tape only).

SEE ALSO

kill(1)
ps(1)
sh(1)
backup(8)

NAME

sync – update the super block

SYNOPSIS

sync

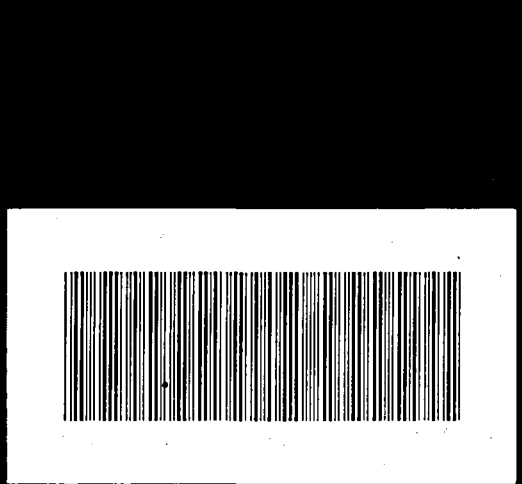
DESCRIPTION

Sync executes the **sync** system primitive. If the system is to be stopped, **sync** must be called to insure file system integrity. See **sync(2)** for details.

SEE ALSO

sync(2)
update(8)

Index



Order Number
DSW-022

Document Number
710-018230-001