# CONVEX SPU UNIX Utilities Manual

Document No. 760-000530-000

Second Edition
March 1990

CONVEX Computer Corporation
Richardson, Texas USA

*CONVEX SPU UNIX Utilities Manual*
Order No. DHW-021
Second Edition



Printed in the United States of America

# Revision Sheet

*CONVEX SPU UNIX Utilities Manual*

| Edition | Document No. | Date | Description |
|---|---|---|---|
| Second | 760-000530-000 | March 1990 | This release contains the new manpage: *kermit.1*. |
| 1.0 | 760-000850-201 | March 1988 | First release, Version 1.0. System call error numbers, UNIX V7 utilities, file formats, and system management utilities are from online manual pages, previously contained in chapters 1, 5, and 8 of the *CONVEX SPU UNIX Programmer's Manual*. |

THIS PAGE INTENTIONALLY LEFT BLANK

# Table of Contents



# Appendixes

# List of Figures

# List of Utilities

## 1. Commands



## 2. System Calls



## 5. File Formats

**8. System Management**

# Permuted Index – Utilities

# Preface

## Purpose and Audience

This document contains the man pages that describe the features of the CONVEX Service Processor Unit (SPU) Operating System utilities.

This manual is designed for CONVEX field and design engineers, manufacturing personnel, and CONVEX customers who perform their own maintenance.

## Scope

The information in this manual applies to any system whose SPU executes SPU UNIX. Currently this includes the CONVEX C1, C120, C201, C202, C210, C220, C230, and C240 supercomputers.

## Outline

This manual is designed to maintain compatibility with the *CONVEX SPU UNIX Programmer's Manual.* As such, each chapter of this manual presents information about certain SPU UNIX features. Because SPU UNIX is not intended as a program development environment for the customer, certain features, such as system calls and library routines are not included in this manual. Hence, Chapters 3, 4, 6, and 7 do not contain any entries. The chapter header is included in this manual only to maintain compatibility with the *CONVEX SPU UNIX Programmer's Manual.*

The *CONVEX SPU UNIX Utilities Manual* comprises eight chapters. The first chapter contains descriptions of the UNIX V7 commands. Chapter 2 gives is a brief introduction to system call error numbers. Chapters 3, 4, 6, and 7 contain no entries; these chapters are simply place holders to maintain format compatibility with the *CONVEX SPU UNIX Programmer's Manual.* Chapter 5 contains SPU UNIX file formats. Chapter 8 contains information related to SPU system operation and maintenance. The utilities in chapters 1, 5, and 8 are in alphabetical order and match the style and format of the standard *CONVEX SPU UNIX Programmer's Manual.*

The content of each chapter is outlined below:

**Chapter 1. UNIX V7 Commands** — This chapter describes publicly-accessible UNIX V7 commands.

**Chapter 2. System Call Error Numbers** — This chapter describes system call error numbers only. System calls are not documented.

**Chapter 3. Library Functions** — Because library functions are not included in SPU UNIX software distribution, this chapter contains no entries. This chapter is added to maintain compatibility with standard UNIX documentation.

**Chapter 4. Special Files** — Because special files, related driver functions, and networking support is not included in SPU UNIX software distribution, this chapter contains no entries. This chapter is added to maintain compatibility with standard UNIX documentation.

**Chapter 5. File Formats** — SPU UNIX file formats are discussed in this chapter.

**Chapter 6. Games** — Because games are not included in SPU UNIX software distribution, this chapter contains no entries. This chapter is added to maintain compatibility with standard UNIX documentation.

**Chapter 7. Miscellaneous Documentation** — This chapter contains no entries. It is added to maintain compatibility with standard UNIX documentation.

**Chapter 8. System Management** — This chapter contains information related to the SPU system operation and maintenance.

**Appendix A. Problem Reporting** — The chapter contains information about how to use the *contact* facility to report problems.

# How To Use This Manual

Only Chapters 1, 2, 5, and 8 contain entries. Chapter 1 contains publicly-accessible UNIX V7 commands; Chapter 2 contains only the introduction to system call error numbers; Chapter 5 contains executable file format files; and Chapter 8 contains system maintenance and operation commands.

All entries are based on a common format, not all of which will always appear:

- The NAME subsection lists the exact names of the commands and subroutines covered under the entry and gives a short description of their purpose.
- The SYNOPSIS summarizes the use of the program being described.
- The DESCRIPTION subsection discusses in detail the subject at hand.
- The FILES subsection gives the names of files that are built into the program.
- A SEE ALSO subsection gives pointers to related information. Each cross-referenced command is boldfaced and contains the filename plus a number in parentheses. The number is parentheses is the chapter number for the cross-reference. Any cross-reference with a (2), (3), (4), (6), or (7) will not be found in this manual.
- A DIAGNOSTICS subsection discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.
- The BUGS subsection gives known bugs and sometimes deficiencies. Occasionally the suggested fix is described.

# Notational Conventions

A few conventions are used, particularly in the "Commands" subsection:

- **Boldface** words are considered literals, and are typed just as they appear.

- Square brackets ( [ ] ) around an argument indicate that the argument is optional. When an argument is given as "name," it always refers to a filename.
- Ellipses ( ... ) are used to show that the previous argument-prototype may be repeated.
- All CONVEX illustrations have an illustration catalog number at the bottom right-hand corner that is for CONVEX use only.

# Associated Documentation

The following is a partial list of other manuals or books that may provide more detailed information on the topics presented in this manual:

- *CONVEX Processor Operation Guide (C100 Series, C200 Series)*, Order No. DHW-015
- *CONVEX Diagnostic Utilities Manual (C1, C120)*, Order No. DHW-072
- *CONVEX Diagnostic Utilities Manual (C200 Series)*, Order No. DHW-082
- *CONVEX UNIX Tutorial Papers*, Order No. DSW-002
- *The C Programming Language*, Kernighan & Ritchie, Order No. DSW-046

# Ordering Documentation

To order the most current version of this or any other CONVEX document, use the 6-digit order number. If the order number is not known, order by the exact title. In some situations, the most current version may not be desired. To receive a specific version of a manual, order the manual by its 12-digit document, or part, number, which can be obtained by contacting the local CONVEX office or the Technical Assistance Center.

The order number for this manual is DHW-021.
The document number for this manual is 760-000530-000.

CONVEX documents can be ordered by mail by sending a request to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

# Technical Assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC):

- From all locations in the continental United States, call 1(800)952-0379
- From locations in Alaska, Hawaii, and Canada, call 1(214)497-4379
- From all other locations, contact the nearest CONVEX office.

# Electronic Mail

The Hardware Documentation Group has an email address for documentation comments. Use this service to give us a quick response mechanism if you have special documentation questions that you would like addressed immediately. If you have a technical question, you should still contact the Technical Assistance Center, as described previously. To use email response service, just send mail addressed to:

```
cnvxhwdoc@convex.COM
```

We will read your comments and give you a personal reply.

# What to Include in an Email Message

When you use the electronic mail service, please provide the following information:

- The reader's name and company name
- A return email address in INTERNET notation or UUCP (bang) notation
- The manual that is being critiqued
- The chapter and page number in question
- The comment

# Reader's Forum

If you wish to mail your comments to us, please use the form at the end of this manual and list the document page number with your questions and comments. Thank you.

# Acknowledgments

I would like to thank the following people for their contributions to this manual:

- Technical contributors: Brian Allison, Craig Reed, and John Sheley
- Document review team: Larry Bonura, Harold Lewis, Craig Reed, and John Sheley
- Hardware documentation staff: Larry Bonura and David Massey

Without the efforts of the aforementioned, this document would not have been possible.

Randall Stiles
CONVEX Hardware Documentation

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 1
# UNIX V7 Commands

## 1.1 Overview

Each CONVEX computer system contains a single-board microcomputer called the Service Processor Unit (SPU). The SPU is responsible for initializing the computer system, booting CONVEX UNIX, monitoring system operation, and executing system diagnostics during maintenance. These functions are performed by SPU-based programs that execute under the SPU operating system known as the SPU UNIX.

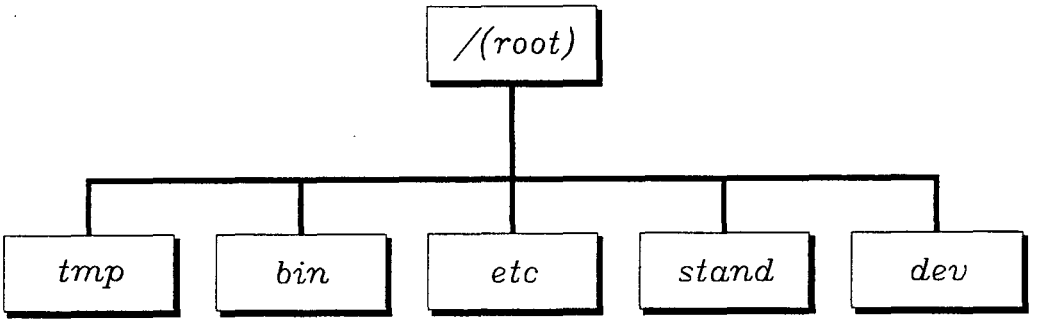
The SPU operating system is based on UNIX Version 7. Although more primitive than CONVEX UNIX, the user interface is recognizable to anyone familiar with the UNIX Bourne shell. Version 7 is a nondemand page version of UNIX, which means that an entire program must be resident in SPU memory before program execution can begin.

This manual documents the standard utilities (commands) available under SPU UNIX. This manual does not describe SPU UNIX system calls or library routines since SPU UNIX is not intended to be a program development environment, and the capability to use these routines is not included in the SPU UNIX release. Additionally, this manual does not document system initialization and diagnostic utilities that have been developed by CONVEX but are not part of the SPU UNIX release. These utilities are documented in the diagnostic utilities manuals for the various CONVEX computer systems.

## 1.2 SPU UNIX Directories

SPU UNIX has the following standard directory structure:

**Figure 1–1, SPU UNIX Directory Structure**



The purpose of each directory is listed below:

- */tmp* — Used as scratch pad and is not backed up
- */bin* — UNIX utilities, e.g. *kermit, more,* and *pwp*
- */etc* — All other UNIX system administration utilities, e.g., *fsck* and *backup*
- */stand* — Standalone test programs, e.g., *spu2000*
- */dev* — Special files for various devices, such as tape drives or disk drives

## 1.3 UNIX V7 Commands

This chapter describes UNIX V7 commands.

NAME

intro – introduction to commands

DESCRIPTION

This section describes publicly accessible commands in alphabetic order.

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.

(*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 *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.

$$b1 \le address < e1 \Rightarrow file\ address = address + f1 - b1, \text{ otherwise,}$$

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

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

cmp – compare two files

SYNOPSIS

**cmp** [ –l] [ –**s**] [ –**x**] 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, *cmp* 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.

**–x** Same as –l except that differing bytes are printed in hex instead of octal.

**–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

In the first synopsis, the **copy** utility copies *file1* 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 filenames.

The **copy** utility refuses to copy a file onto itself.

SEE ALSO

**cat(1)**
**mv(1)**

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 name should be placed in the file */etc/timezone* so that the time zone will be properly set at boot time.

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
*/etc/timezone* time zone name used at boot time

SEE ALSO

**ctime(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

The **dd** utility copies the specified input file to the specified output with possible conversions. The standard Input and Output (I/O) are used by default. The I/O 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 November 1968.

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.

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

kermit – kermit file transfer

SYNOPSIS

**kermit** [ option ...] [file ...]

DESCRIPTION

**Kermit** is a file transfer program that allows files to be moved between machines of many different operating systems and architectures. This man page describes version 4E(068) of the program. Refer to the C-Kermit documentation (ckuker.doc) or the Kermit book for further information.

Arguments are optional. If **kermit** is executed without arguments, it will enter interactive command mode. Otherwise, **kermit** will read the arguments off the command line and interpret them.

The following notation is used in command descriptions:

*fn* A UNIX file specification, possibly containing either of the wildcard characters (an asterick [*] which matches all character strings or a question mark [?] which matches any single character).

*fn1* A UNIX file specification which may not contain a wildcard character.

*rfn* A remote file specification in the remote system's own syntax, which may denote a single file or a group of files.

*rfn1* A remote file specification which should denote only a single file.

*n* A decimal number, in most cases between 0 and 94.

*c* A decimal number between 0 and 127 representing the value of an ASCII character.

*cc* A decimal number between 0 and 31, or else exactly 127, representing the value of an ASCII control character.

[ ] Any field in square braces is optional.

{*x,y,z*} Alternatives are listed in curly braces.

COMMAND LINE OPTIONS

**Kermit** command line options may specify either actions or settings. Action options specify either protocol transactions or terminal connection.

**–s** *fn* Send the specified file or files. If *fn* contains wildcard characters, the UNIX shell expands it into a list. If *fn* is a dash [-] then **kermit** sends from standard input, which may come from a file:

```
kermit -s - < foo.bar
```

or a parallel process (cannot be used to send terminal typein):

```
ls -l | kermit -s -
```

To send a file whose name is a dash, a pathname can be used to precede the dash:

```
kermit -s ./-
```

**–r** Receive a file or files. Wait passively for files to arrive.

**–k** Receive (passively) a file or files, sending them to standard output. This option can be used in several ways. To display the incoming files on your screen (local mode only):

```
kermit -k
```

To send the incoming file or files to the named file, *fn1*. If more than one file arrives, all are concatenated together into the single file *fn1*.

```
kermit -k > fn1
```

To pipe the incoming data (single or multiple files) to the indicated command:

```
kermit -k | sort > sorted.stuff
```

**–a** *fn1* If a file transfer option is specified, an alternate name for a single file may be specified with the **-a** option. To send the file foo telling the receiver that its name is bar:

```
kermit -s foo -a bar
```

If more than one file arrives or is sent, only the first file is affected by the **–a** option. To store the first incoming file under the name baz:

```
kermit -ra baz
```

**–x** Begin server operation. May be used in either local or remote mode.

**Kermit** is local if it is running on a PC or workstation that is being used directly, or if is running on a multiuser system and transferring files over an external communication line (not the user's job's controlling terminal or console). **Kermit** is remote if it is running on a multiuser system and transferring files over its own controlling terminal's communication line, connected to the user's PC or workstation.

On most systems **kermit** runs in remote mode by default, so on a PC or workstation, **kermit** will have to be put into local mode. The following command sets **kermit's** mode:

**–l** *dev* Line — Specify a terminal line to use for file transfer and terminal connection:

```
kermit -l /dev/ttyi5
```

When an external line is being used, some additional options may be needed for successful communication with a remote system:

**–b** *n* Baud — Specify the baud rate for the line given in the **–l** option. This option should always be included with the **–l** option, since the speed of an external line is not necessarily the speed expected.

```
kermit -l /dev/ttyi5 -b 9600
```

**–p** *x* Parity — **e**, **o**, **m**, **s**, **n** (even, odd, mark, space, or none). If parity is other than none, then the 8th-bit prefixing mechanism will be used for transferring 8-bit binary data, provided the opposite **kermit** agrees. The default parity is none.

**–t** Specifies half duplex, line turnaround with XON as the handshake character.

The following commands may be used only when **kermit** is local — either by default or because the **–l** option has been specified:

**–g** *rfn* Actively request a remote server to send the named file or files; *rfn* is a file specification in the remote host's own syntax. If *fn* contains any special shell characters, like *, these characters must be quoted:

```
kermit -g x\*.\?
```

**–f** Send a finish command to a remote server.

**–c** Establish a terminal connection over the specified or default communication line before any protocol transaction takes place. To return to the local system, type the escape character (normally Control-Backslash) followed by the letter c.

**–n** Like **–c,** but after a protocol transaction takes place; **–c** and **–n** may both be used in the same command.

On a timesharing system, the **–l** and **–b** options also have to be included with the **–s**, **–r**, or **–k** options if the other **kermit** is on a remote system.

If **kermit** is in local mode, the screen (standard output) is continuously updated to show the progress of the file transer. A dot is printed for every four data packets, other packets are shown by type (S for Send-Init, T for a timeout, and % for each retransmission). In addition, certain interrupt commands may be typed (standard input) during file transfer:

**^F** (Control-F) — Interrupt the current file, and go on to the next (if any).

**^B** (Control-B) — Interrupt the entire batch of files, terminate the transaction.

**^R** (Control-R) — Resend the current packet.

**^A** (Control-A) — Display a status report for the current transaction.

These interrupt characters differ from the ones used in other **kermit** implementations to avoid conflict with UNIX shell interrupt characters. With System III and System V implementations of UNIX, interrupt commands must be preceeded by the escape character (e.g., Control-Backslash).

Several other command-line options are provided:

**–i** Specifies that files should be sent or received exactly as is with no conversions. This option is necessary for transmitting binary files. It may also be used to slightly boost efficiency in UNIX-to-UNIX transfers of text files by eliminating CRLF/newline conversion.

**–e** *n* Specifies the extended receive-packet length. The length can be a number between 10 and about 1000 (depending on the system). Lengths of 95 or greater require that the opposite **kermit** support the long packet protocol extension.

**–w** Write-Protect — Avoid filename collisions for incoming files.

**–q** Quiet — Suppress screen update during file transfer, for instance to allow a file transfer to proceed in the background.

**–d** Debug — Record debugging information in the file debug.log in the current directory. Use this option if the program is not executing normally and show the resulting log to the local **kermit** maintainer.

**–h** Help — Display a brief synopsis of the command line options.

The command line may contain no more than one protocol action option.

## INTERACTIVE OPERATION

**Kermit's** interactive command prompt is C-Kermit>. Any valid command may be entered at this prompt. **Kermit** executes the command and then the prompt is redisplayed when **Kermit** is ready for another command. This process continues until the program is terminated.

Commands begin with a keyword, normally an English verb, such as send. Trailing characters may be omitted from any keyword, so long as sufficient characters are entered to distinguish the keyword from any other keyword valid for that field. Certain commonly used keywords such as send, receive, and connect have special non-unique abbreviations (s for send, r for receive, c for connect).

Certain characters have special functions in interactive commands:

**?** (Question mark) — May be typed at any point in a command to display a message explaining what is possible or expected at that point. Depending on the context, the message may be a brief phrase, a menu of keywords, or a list of files.

**ESC** (The Escape or Alt mode key) — Request completion of the current keyword or filename, or insertion of a default value. The result will be a beep if the requested operation fails.

**DEL** (The Delete or Rubout key) — Delete the previous character from the command. **BS** (Backspace, Control-H) may also be used for this function.

**^W** (Control-W) — Erase the rightmost word from the command line.

**^U** (Control-U) — Erase the entire command.

**^R** (Control-R) — Redisplay the current command.

**SP** (Space) — Delimits fields (keywords, filenames, numbers) within a command. **HT** (Horizontal Tab) may also be used for this purpose.

**CR** (Carriage Return) — Enters the command for execution. **LF** (Linefeed) or **FF** (Formfeed) may also be used for this purpose.

**** (Backslash) — Enter any of the above characters into the command, literally. To enter a backslash, type two backslashes in a row (\\). A single backslash immediately preceding a carriage return continues the command on the next line.

The editing characters (DEL, ^W, etc.) may be typed repeatedly to delete all characters back to the prompt. No action will be performed until the command is activated by entering a carriage return, linefeed, or formfeed. If an error is entered, an informative error message and a new prompt is displayed.

Interactive **kermit** accepts commands from files as well as from the keyboard. Upon startup, **kermit** looks for the file .kermrc in the home directory and then in the current directory and executes any commands in the file. These commands must be in interactive format, not UNIX command-line format. The **take** command is also provided for use at any time during an interactive session. Command files may be nested.

The **kermit** interactive commands include the following:

**! command** Execute a UNIX shell command. A space is required after the !.

**% text** A comment. Useful in take-command files.

**bye** Terminate and log out a remote **kermit** server.

**close** Close a log file.

**connect** Establish a terminal connection to a remote system.

**cwd** Change working directory.

**dial** Dial a telephone number.

**directory** Display a directory listing.

**echo** Display arguments literally. Useful in take-command files.

**exit** Exit from the program, closing any open logs.

**finish** Instruct a remote **kermit** server to exit, but not log out.

**get** Get files from a remote **kermit** server.

**help** Display a help message for a given command.

**log** Open a log file — debugging, packet, session, transaction.

**quit** Same as the **exit** command.

**receive** Passively wait for files to arrive.

**remote** Issue file management commands to a remote **kermit** server.

**script** Execute a login script with a remote system.

**send** Send files.

**server** Begin server operation.

**set** Set various parameters.

**show** Display values of **set** parameters, program version, etc.

**space** Display current disk space usage.

**statistics** Display statistics about most recent transaction.

**take** Execute commands from a file.

The **set** parameters include the following:

**block-check** Level of packet error detection.

**delay** How long to wait before sending first packet.

**duplex** Specify which side echoes during connect.

**escape-character** Character to prefix escape commands during connect.

**file** Set various file parameters.

**flow-control** Communication line full-duplex flow control.

**handshake** Communication line half-duplex turnaround character.

**line** Communication line device name.

**modem-dialer** Type of modem-dialer on communication line.

**parity** Communication line character parity.

**prompt** Change the **kermit** programs prompt.

**receive** Set various parameters for inbound packets.

**retry** Set the packet retransmission limit.

**send** Set various parameters for outbound packets.

**speed** Communication line speed.

The **remote** commands include the following:

**cwd** Change remote working directory.

**delete** Delete remote files.

**directory** Display a listing of remote file names.

**help** Request help from a remote server.

**host** Issue a command to the remote host in its own command language.

**space** Display current disk space usage on remote system.

**type** Display a remote file on your screen.

**who** Display the users currently logged in, or get information about a user.

FILES

| | |
|---|---|
| $HOME/.kermrc | *Kermit* initialization commands |
| ./.kermrc | more *Kermit* initialization commands |

SEE ALSO

cu(1C), uucp(1C)
Frank da Cruz, *Kermit User's Guide*, Columbia University, 6th Edition
Frank da Cruz, *Kermit, A File Transfer Protocol*, Digital Press (1987)
The file ckuker.doc.

DIAGNOSTICS

The diagnostics produced by **kermit** are intended to be self-explanatory.

BUGS

See recent issues of the Info-Kermit digest (on ARPANET or Usenet), or the file ckuker.bwr, for a list of bugs.

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

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, lf, ll, lr – list contents of directory

SYNOPSIS

**ls** [ **–acdfgilqrstu1ACLFR** ] name ...
**lf** [ **–acdfgilqrstu1ACLR** ] name ...
**ll** [ **–acdfgiqrstu1ACLFR** ] name ...
**lr** [ **–acdfgilqrstu1ACLF** ] 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.

There are three common options that have built-in abbreviations: **lf** is equivalent to **ls -F**; **ll** is interpreted as **ls -l**; and **lr** will display the same information as **ls -R**. Each of these special forms will also take any of the additional options listed below.

By default, the output is sorted alphabetically. 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 a large number of 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. If the file is a symbolic link, the pathname of the linked-to file is printed preceded by "->".

**–g** Include the group ownership of the file in a long output.

**–t** Sort by time modified (latest first) instead of by name.

**–a** List all entries. In the absence of this option, entries whose names begin with a period ( . ) are *not* listed.

**–s** Give size in kilobytes of each file.

**–d** If argument is a directory, list only its name. Often used used with **-l** to get the status of a directory.

**–L** If argument is a symbolic link, list the file or directory the link references rather than the link itself.

**–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 (with the **-t** option) or printing (with the **-l** option).

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

**–i** For each file, print the i-number in the first column of the report.

**–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.

**–A** If argument is a directory, list its content.

**–F** cause directories to be marked with a trailing '/', sockets with a trailing "=", symbolic links with a trailing "@", and executable files with a trailing '*'.

**–R** recursively list subdirectories encountered.

**–1** force one entry per line output format; this is the default when output is not to a terminal.

**–C** force multi-column output; this is the default when output is to a terminal.

**-q** force printing of nongraphic characters in file names as the character '?'; this is the default when output is to a terminal.

The mode printed under the **-l** option contains 10 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
**l** if the entry is a symbolic link
**p** if the entry is a FIFO (a.k.a. "named pipe") special file
**s** if the entry is a socket
– if the entry is a plain file

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

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

## FILES

*/etc/passwd* to get user IDs for **ls -l**.
*/etc/group* to get group IDs 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 that used **ls** almost certain losers.

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

The **mv** utility 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.

The **mv** utility 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.

It is an historical botch to require specification of object, radix, and sign representation in a single character argument.

NAME

proctype – tell what type of SPU is being run

SYNOPSIS

**proctype**

DESCRIPTION

The **proctype** utility exits with the value returned by the *proc_type* system call. Currently the only defined values are 1 for SP1 and 2 for SP2.

SEE ALSO

**proc_type(2)**

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

The **reset** utility sets the terminal to cooked mode, turns off *cbreak* and *raw* modes, turns on *nl*, and restores undefined special characters to their default values.

The **reset console** option affects both the local console and the modem port, and should be used in most cases.

The **reset tty** option is useful in multi-user mode, and affects only the user's terminal.

The **reset** utility 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

Does not set tabs properly; it cannot 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** [ **–mubdfinrt***x* ] [ +*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 default tape drive (archive tape is */dev/rmt0*; cipher tape is */dev/rct0e*). 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.

SEE ALSO

**ct(4)** for Cipher tape drive (for C1 only)
**dk(4)**
**mtio(4)** for QIC tape drive (for C1, C120, C210, or C220)

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

The **test** utility evaluates the expression *expr*. If its value is true, a zero exit status is returned; otherwise, a nonzero exit status is returned. If there are no arguments, **test** returns a nonzero exit.

To allow certain constructs in **sh** scripts, **test** may also be invoked as [. In this case, a closing ] must terminate the expression; otherwise, **test** will generate an error message. An example of using **test** with [ within a **sh** script is:

**if** [ *expr* ]

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**. In order to ensure correct parsing, all arguments should be separated by blanks. 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

vi – screen oriented (visual) text editors based on ex

SYNOPSIS

**vi** [ **–t** tag ] [ **–r** ] [ **–R** ] [ *+command* ] [ **–l** ] [ **–w***n* ] [ **–x** ] name ...

DESCRIPTION

The **vi** (visual) utility is a display oriented text editor based on **ex**. Since **ex** and **vi** run the same code, it is possible to get to the command mode of **ex** from within **vi** and vice-versa. The use of **vi** is described in the *CONVEX UNIX Tutorial Papers* and in the *CONVEX UNIX Text Editor User's Guide.*

Options available with **vi** are:

| | |
|---|---|
| **–t** | Edit the file containing the *tag* and position the editor at its definition. |
| **–R** | Set *readonly* option which is used to read but not modify a file. |
| *+command* | Editor begins execution by processing the specified command. |
| **–l** | Sets the *showmatch* option. |
| **-w***n* | Set the default window size to *n*. |

SEE ALSO

*CONVEX UNIX Text Editor User's Guide*
"An Introduction to Display Editing with Vi (Revised)" in the *CONVEX UNIX Tutorial Papers*
**ex (1)**

BUGS

Software tabs using **^T** work only immediately after the *autoindent.*

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The *wrapmargin* option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line will not be broken.

Inserting or deleting within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Saving text on deletes in the buffers is inefficient.

Exiting **vi** or **ex** with **ZZ** or **:q** immediately after executing **:1,$d** results in no changes to the file, i.e., no lines of the file will be changed or modified in the editing session.

The **source** command does not work when executed as **:source**; there is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a **:** escape. To use these on a **:global** you must **Q** to **ex** command mode, execute them, and then re-enter the screen editor with **vi** or **open.**

Using "w" to jump over the last word on a line will not work if the line ends in a blank.

RESTRICTIONS

The "_" key by itself should not be mapped to anything else.

NAME

which – locate all executable occurrences of a program file in $PATH

SYNOPSIS

**which name** [ ... ]

DESCRIPTION

The **which** utility takes a list of names and looks for all executable occurrences of those files in each path in the environment variable $PATH. The first occurrence found is always the file that would be executed had **name** been given as a command.

SEE ALSO

**sh(1)**

BUGS

The **which** utility does not recognize **sh** built-in functions such as **cd**, **for**, etc.

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.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 2
# System Call Error Numbers

## 2.1 Overview

This chapter contains only the introduction to system call error numbers. SPU UNIX system calls are not documented since the capability to develop SPU-based programs is not included in the SPU UNIX release.

NAME

intro, errno – introduction to system call error numbers

DESCRIPTION

Most UNIX V7 system calls have an error return. Here is a list of the error numbers and the standard UNIX V7 error messages:

0 Error 0
Unused.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
The process whose number was given to *signal* and *ptrace* does not exist, or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error occurred during a *read* or *write*. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice that does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialled in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5120 bytes is presented to *exec*.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see **b.out(5)**.

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file that is open only for writing (resp. reading).

10 ECHILD No children
*Wait* and the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a *fork*, the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough core
During an *exec* or *break*, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g. in *mount*.

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment).

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g. *link*.

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir*.

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in *signal*, reading or writing a file for which *seek* has generated a negative pointer. Also set by math functions (see **intro(3)**).

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in *stty* or *gtty* is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program that is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about $10^9$ bytes).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek* was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 links to a file.

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

**SEE ALSO**

**intro(3)**

# Chapter 3
# Library Functions

## 3.1 Overview

This chapter is intentionally left blank. SPU UNIX library routines are not documented since the capability to develop SPU-based programs is not included in the SPU UNIX release. The chapter header is included here to maintain compatibility with the *CONVEX SPU UNIX Programmer's Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 4
# Special Files

## 4.1 Overview

This chapter is intentionally left blank. Special files, related driver functions, and networking support are not documented since the capability to develop SPU-based programs is not included in the SPU UNIX release. The chapter header is included here to maintain compatibility with the *CONVEX SPU UNIX Programmer's Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 5
# File Formats

## 5.1 Overview

This chapter contains information related to SPU UNIX file formats.

THIS PAGE INTENTIONALLY LEFT BLANK

NAME

b.out – executable file format

DESCRIPTION

The assembler *(a68)* and linker *(ld68)* produce executable files in this format. The system call *exec* expects files of this format.

**B.out** has six sections:

1. header
2. program text
3. data text
4. symbol table
5. text relocation commands
6. data relocation commands

These sections are always in the above order.

The header always contains the following 8 longwords:

1. magic number (default is 407)
2. the size of the program text segment
3. the size of the initialized portion of the data segment
4. the size of the uninitialized (bss) portion of the data segment
5. the size of the symbol table
6. the size of the text relocation commands
7. the size of the data relocation commands
8. the entry location for program execution

The sizes of all **b.out** segments are in bytes. The sizes of the text, data, and bss segments are rounded multiples of 4, adjusting to the next longword boundary. Starting points for all other segments depend upon the size of the preceeding segments. Listed below are the abbreviations used, for clarity, in discussing the starting points for **b.out** segments.

| | |
|---|---|
| st | size of the text segment |
| sd | size of the data segment |
| ss | size of the symbol table |
| str | size of the text relocation segment |

The starting positions, in bytes, for **b.out** segments are:

| | |
|---|---|
| text segment | 32 |
| data segment | 32 + st |
| symbol table | 32 + st + sd |
| text relocation | 32 + st + sd + ss |
| data relocation | 32 + st + sd + ss + str |

Three logical segments are set up when a file produced by *ld68* is loaded into core for execution on the SP2:

1. text segment - loaded starting at 0x80000.
2. data segment - divided into *data* and *bss*. The *data* part contains initialized data. The *bss* contains uninitialized data which starts off as all 0 upon execution. The data segment may be extended by using the *brk* system call.
3. stack - occupies the highest possible location in the core image. The stack grows down from 0xfffff.

The symbol table consists of variable length entries. The first byte contains the symbol type, the next four bytes comprize a longword which contains the value of the symbol, and the remaining bytes contain the symbol name (zero-padded). The possible symbol types are:

00 undefined symbol
01 absolute symbol
02 text segment symbol
03 data segment symbol
04 bss segment symbol
06 register symbol
040 undefined external (.globl) symbol
041 absolute external symbol
042 text segment external symbol
043 data segment external symbol
044 bss segment external symbol

*NOTE: 040 is ORed (40 | n) to symbol types 00-04 to identify corresponding external symbols. Ld68 interprets type 40 (undefined external) symbols with non-zero values as common regions with a size equal to the symbol value.*

The loader, *ld68*, defines information which is undefined prior to the load, and adjusts data after linking several files together. Information from the loader is required when a word is to be relocated, or when a relocation command refers to an undefined external symbol. When files are linked, all undefined external symbols are defined, and the values of the symbols are added into the correct words in the output. Words which are to be relocated, but do not involve external references, contain the value which should be present if the file is executed without linking. If linking modifies the word, it should be adjusted by adding the new address of the beginning of the region in which it was relocated.

Relocation information consists of a sequence of commands which specify how to fix up bytes in the text and data portion of the file. Relocation information comes in two parts: commands for the text segment, and commands for the data segment. Each command has four parts *(NOTE: Bit 0 is the rightmost bit in the byte):*

1. Bits 7-6 of the first byte specify which segment this relocation command refers to:
   00 add the address of the start of the text segment
   01 add the address of the start of initialized data
   02 add the address of the start of bss (uninitialized data)
   03 the value of the external symbol is put into the word
2. Bits 5-4 of the first byte indicate the number of bytes to be relocated:
   00 byte
   01 word
   02 long
3. The second word in the relocation command is used for undefined external symbols. All undefined external symbols are read into a symbol array. A C-style index is used. The value of the word is the index into the symbol array.
4. A longword is given which indicates the position of the data to be modified. This position is relative to the start of the segment, *not to the start of the file.* Thus, a command to relocate the first byte of the data segment would have a position of zero.

The following is the C structure for the relocation information:

```
struct reloc {
        short rinfo;            /* rsegment, rsize, and rdisp */
/*      char rsegment:2;        /* RTEXT, RDATA, RBSS, or REXTERN */
/*      char rsize:2;           /* RBYTE, RWORD, or RLONG */
/*      char rdisp:1;           /* 1 => a displacement */
        short rsymbol;          /* ID of the symbol of external relocations */
        long rpos;              /* position of relocation in segment */
};
```

The following is the C structure for the symbol table information:

```
struct  nlist {                 /* symbol table entry */
        char n_name[8];         /* symbol name */
        int n_type;             /* type flag */
        unsigned n_value;       /* value */
};
```

SEE ALSO

HOST UNIX: **a68(1)**(optional product), **ld68(1)**(optional product)
SP2 UNIX: **exec(2)**, **brk(2)**, **nlist(3)**

NAME

backup – backup tape format

DESCRIPTION

This man page describes the data format developed to allow machine-specific versions of a released product and/or different products to be distributed on a single cartridge tape, and to increase flexibility of bootimage tapes to allow for future expansion. This format applies only to the QIC-compatible tape; this format is not supported for the Cipher floppy tape drive. Data structures that define this format can be found in the 68k include file <sys/backup.h>.

FORMAT

The beginning of the tape contains a header of the following format:

```
#define TA_MAGIC 0x6000621
#define TA_MAXENTRIES 47
#define TA_SP1 1
#define TA_SP2 2
#define TA_SPFUTURE 4
struct ta_header
{
    unsigned long magic_number; /* TA_MAGIC */
    char date[32];
    char creator[16];
    char reserved[30];
    struct ta_fileinfo fileinfo[TA_MAXENTRIES];
}

struct ta_fileinfo
{
    char product[16];
    unsigned short machine;      /* TA_SP1, TA_SP2, etc... */
    unsigned short filemarks;    /* number to space to find
                                        fileheader structure */
}
```

This structure allows anythings that needs to find a section of the tape to look through the fileinfo array for a string and machine code matching the desired data. The number contained in the filemarks field is the number of "file skip forward" commands needed to reach the data. In front of each block of data is another structure of the form:

```
struct ta_fileheader
{
    char product[16];
    unsigned short machine;              /* TA_SP1, TA_SP2, etc... */
    unsigned short num_copies;
    unsigned long bytes_per_copy;
    unsigned short filemarks;
}
```

Here the product and machine info should be identical to that in the master header (it should be considered an error if this is not so). Num_copies is how many copies of the data are contained in the section, bytes_per_copy is the size of each copy of the data, and filemarks is how many filemarks are intersperced through the data. Normally this field will be 1 indicating there is only a filemark after the concatenation of all copies, but can be more if it is desired to separate copies by filemarks. SPU EPROMS, boot tracks, *installsw(8)*, and other software will use this information to locate the appropriate data on the tape. The following is an example format for a

combination SP1/SP2 bootimage and installsw tape.

EXAMPLE TAPE FORMAT

```
Tape header
   magic_number (0x6000621)
   char date ("Thu Sep 10 15:02:52 CDT 1987")
   char creator ("/etc/backup")
   char reserved[30]
   struct ta_fileinfo fileinfo[TA_MAXENTRIES]
      product ("boot tracks")
      machine (TA_SP1)
      filemarks (1)
      product ("boot tracks")
      machine (SP2)
      filemarks (10)
      product ("installsw")
      machine (TA_SP1 | TA_SP2)  # applies to both machine types
      filemarks (20)
        (all other entries empty)
(filemark)
# master header entry for "boot tracks" (sp1) points here
file header
   product ("boot tracks")
   machine (SP1)
   num_copies (4)
   bytes_per_copy (16384)
   filemarks (1)
(filemark)
(4 copies of boot track for sp1)
(filemark)
file header
   product ("spu2000")
   machine (SP1)
   num_copies (2)
   bytes_per_copy (52224)
   filemarks (1)
(filemark)
(2 copies of spu2000)
(filemark)
file header
   product ("backup info")
   machine (SP1)
   num_copies (1)
   bytes_per_copy (512)
   filemarks (1)
(filemark)
(1 copy of backup information)
(filemark)
file header
   product ("root partition")
   machine (SP1)
   num_copies (2)
   bytes_per_copy (2045952)
   filemarks (2)
```

```
(filemark)
(copy 1 of root partition)
(filemark)
(copy 2 of root partition)
(filemark)
# master header entry for "boot tracks" (sp2) points here
file header
   product ("boot tracks")
   machine (SP2)
   num_copies (4)
   bytes_per_copy (16384)
   filemarks (1)
(filemark)
(4 copies of boot track for sp2)
(filemark)
file header
   product ("spu2000")
   machine (SP2)
   num_copies (2)
   bytes_per_copy (52224)
   filemarks (1)
(filemark)
(2 copies of spu2000)
(filemark)
file header
   product ("backup info")
   machine (SP2)
   num_copies (1)
   bytes_per_copy (512)
   filemarks (1)
(filemark)
(1 copy of backup information)
(filemark)
file header
   product ("root partition")
   machine (SP2)
   num_copies (2)
   bytes_per_copy (4091904)
   filemarks (2)
(filemark)
(copy 1 of root partition)
(filemark)
(copy 2 of root partition)
(filemark)
# master header entry for "installsw" (sp1 & sp2) points here
(installsw data)
```

SEE ALSO

**backup(8), tar(1), installsw(8), mtio(4), mt(1)**

NAME

core – format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See **signal(2)** for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system's per-user data for the process, including the registers as they were at the time of the fault; see the system listings for the format of this area. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

In general the debugger **adb(1)** is sufficient to deal with core images.

SEE ALSO

**adb(1)**
**signal(2)**

NAME

dir – format of directories

SYNOPSIS

**#include <sys/dir.h>**

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry see, **filsys(5)**. The structure of a directory entry as given in the include file is:

```
/*      $CHeader: dir.h 1.1 87/08/18 13:26:20 $*/
/*      Copyright 1984 Convex Computer Corp.*/

#ifndef DIRSIZ
#define DIRSIZ14
#endif
struct  direct
{
        ino_t   d_ino;
        char    d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases '..' has the same meaning as '.'.

SEE ALSO

**filsys(5)**

NAME

environ – user environment

SYNOPSIS

**extern char **environ;**

DESCRIPTION

An array of strings called the 'environment' is made available by **exec(2)** when a process begins. By convention these strings have the form 'name=value'. The following names are used by various commands:

PATH The sequence of directory prefixes that *sh, time,* **nice(1)**, etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'.

HOME A user's login directory, set by */.profile.*

TERM The kind of terminal for which output is to be prepared.

Further names may be placed in the environment by the *export* command and 'name=value' arguments in **sh(1)**, or by **exec(2)**. It is unwise to conflict with certain Shell variables that are frequently exported by '.profile' files: MAIL, PS1, PS2, IFS.

SEE ALSO

**exec(2)**
**sh(1)**

NAME

filsys, flblk, ino – format of file system volume

SYNOPSIS

**#include <sys/types.h>**
**#include <sys/flbk.h>**
**#include <sys/filsys.h>**
**#include <sys/ino.h>**

DESCRIPTION

Every file system storage volume (e.g. dk disk, cartridge tape) has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the *super block*. The layout of the super block as defined by the include file *<sys/filsys.h>* is:

```
/*      $CHeader: filsys.h 1.1 87/08/18 13:26:25 $*/
/*      Copyright 1984 Convex Computer Corp.*/

/*
 * Structure of the super-block
 */
struct filsys {
        unsigned short s_isize;         /* size in blocks of i-list */
        daddr_t  s_fsize;               /* size in blocks of entire volume */
        short    s_nfree;               /* number of addresses in s_free */
        daddr_t  s_free[NICFREE];/* free block list */
        short    s_ninode;              /* number of i-nodes in s_inode */
        ino_t    s_inode[NICINOD];/* free i-node list */
        char     s_flock;               /* lock during free list manipulation */
        char     s_ilock;               /* lock during i-list manipulation */
        char     s_fmod;                /* super block modified flag */
        char     s_ronly;               /* mounted read-only flag */
        time_t   s_time;                /* last super block update */
        /* remainder not maintained by this version of the system */
        daddr_t  s_tfree;               /* total free blocks*/
        ino_t    s_tinode;              /* total free inodes */
        short    s_m;                   /* interleave factor */
        short    s_n;                   /* " " */
        char     s_fname[6];            /* file system name */
        char     s_fpack[6];            /* file system pack name */
};
```

*S_isize* is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus is i-list is *s_isize*–2 blocks long. *S_fsize* is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free[1], ... , s_free[s_nfree–1],* up to NICFREE free block numbers. NICFREE is a configuration constant. *S_free[0]* is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file *<sys/fblk.h>* is:

```
/*      $CHeader: fblk.h 1.1 87/08/18 13:26:24 $*/
/*      Copyright 1984 Convex Computer Corp.*/

struct fblk
{
        short     df_nfree;
        daddr_t   df_free[NICFREE];
};
```

The fields *df_nfree* and *df_free* in a free block are used exactly like *s_nfree* and *s_free* in the super block. To allocate a block: decrement *s_nfree*, and the new block number is *s_free[s_nfree]*. If the new block address is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read the new block into *s_nfree* and *s_free*. To free a block, check if *s_nfree* is NICFREE; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free[s_nfree]* to the freed block's address and increment *s_nfree*.

*S_ninode* is the number of free i-numbers in the *s_inode* array. To allocate an i-node: if *s_ninode* is greater than 0, decrement it and return *s_inode[s_ninode]*. If it was 0, read the i-list and place the numbers of all free inodes (up to NICINOD) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than NICINODE, place its number into *s_inode[s_ninode]* and increment *s_ninode*. If *s_ninode* is already NICINODE, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

*S_flock* and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information. *S_ronly* is a write-protection indicator; its disk value is also immaterial.

*S_time* is the last time the super-block of the file system was changed. During a reboot, *s_time* of the super-block for the root file system is used to set the system's idea of the time.

The fields *s_tfree, s_tinode, s_fname* and *s_fpack* are not currently maintained.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-nodes are 64 bytes long, so 8 of them fit into a block. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node as given in the include file *<sys/ino.h>* is:

```
/*      $CHeader: ino.h 1.1 87/08/18 13:26:27 $*/
/*      Copyright 1984 Convex Computer Corp.*/

/*
 * Inode structure as it appears on
 * a disk block.
 */
struct dinode
{
        unsigned short      di_mode;    /* mode and type of file */
        short   di_nlink;   /* number of links to file */
        short   di_uid;     /* owner's user id */
        short   di_gid;     /* owner's group id */
        off_t   di_size;    /* number of bytes in file */
        char    di_addr[40]; /* disk block addresses */
        time_t  di_atime;   /* time last accessed */
        time_t  di_mtime;   /* time last modified */
        time_t  di_ctime;   /* time created */
```

```
};

/*
 * Of the 40 address bytes, 39 are used to form 13 addresses (24 bits),
 *      the last is unused.
 */
```

*Di_mode* tells the kind of file; it is encoded identically to the *st_mode* field of **stat(2)**. *Di_nlink* is the number of directory entries (links) that refer to this i-node. *Di_uid* and *di_gid* are the owner's user and group IDs. *Size* is the number of bytes in the file. *Di_atime* and *di_mtime* are the times of last access and modification of the file contents (read, write or create) (see **times(2)**); *Di_ctime* records the time of last modification to the inode or to the file, and is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the *di_addr* field is occupied by the device code (see **types(5)**). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array *di_addr* packed into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresses are singly, doubly, and triply indirect and point to blocks of 128 block pointers. Pointers in indirect blocks have the type *daddr_t* (see **types(5)**).

For block *b* in a file to exist, it is not necessary that all blocks less than *b* exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

**dir(5)**
**fsck(1)**
**mount(1)**
**types(5)**

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

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

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***x*.

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)**
**tset(1)**
**more(1)**

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)**

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)**

BUGS

Some lines are merely known as "dialup" or "plugboard".

# Chapter 6
# Games

## 6.1 Overview

This chapter is intentionally left blank. There are no games distributed with the SPU UNIX release. The chapter header is included here to maintain compatibility with the *CONVEX SPU UNIX Programmer's Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 7
# Miscellaneous Documentation

## 7.1 Overview

This chapter is intentionally left blank. Information normally found in this chapter is not applicable to the SPU UNIX release. The chapter header is included here to maintain compatibility with the *CONVEX SPU UNIX Programmer's Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 8
# System Management

## 8.1 Overview

This chapter contains information related to SPU system operation and maintenance.

THIS PAGE INTENTIONALLY LEFT BLANK

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 (C1 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 (C1 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 (C1 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 (C1 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, C1 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)**

DIAGNOSTICS

**Bldboot** enables the write verify mode in the SPU UNIX cartridge tape driver. This mode will cause the tape to write a block, rewind to the front of the block and then read the block just written. The write verify mode attempts to ensure that the tape written can be read with out errors.

BUGS

Backups on different versions of SPU UNIX may not be compatible. The boot tracks and the root partition are backed up in binary format.

## NAME

bootchk – print SPU UNIX boot time info

## SYNOPSIS

**/etc/bootchk**

## DESCRIPTION

**Bootchk** is used to retrieve the AC line frequency and SPU UNIX boot time from kernel memory. This utility reads the SPU UNIX object file and parses the symbol table for **Hz** and **boottm**. These are used to locate the AC line frequency and boot time respectively. **Bootchk** is usually used in the */.profile* script.

## FILES

| | |
|---|---|
| */unix* | SPU UNIX object file |
| */dev/kmem* | SPU UNIX kernel memory |

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 SPU, disabling 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.

For information on formatting the SPU disk, see the installation procedures on the notes released with the current version of the SPU UNIX tape.

NAME

fsck – file system consistency check and interactive repair

SYNOPSIS

**/etc/fsck** **–p** [ filesystem ... ]
**/etc/fsck** [ **–y** ] [ **–n** ] [ **–sX** ] [ **–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/.profile* during automatic reboot. In this case **fsck** reads the table */etc/fstab* to determine which file systems to check. 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.

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.

**–s***X* 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, incore 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:

–sBlocks-per-cylinder:Blocks-to-skip (for anything else)

If *X* is not given, the values used when the filesystem was created are 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. If the shell terminates, **init** comes up multi-user and the process described below is started.

When **init** comes up multiuser, it invokes a shell, with input taken from the file */etc/.profile.* This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then **init** reads the file */etc/ttys* and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input, output and error files. Opening the typewriter will usually involve a delay, since the *open* is not completed until someone is dialed up and carrier established on the channel. Then */etc/getty* is called with argument as specified by the last character of the *ttys* file line. *Getty* reads the user's name and invokes **login(1)** to log in the user and execute the shell.

Ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up.

**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?*

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

The **installsw** utility is used to create and install C1, C120, C210, C220, and SPU software release tapes. The **installsw** utility may be run on the C1, C120, C210, or C220, 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, 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. 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 system 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 a system, 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 system, 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 system, or cartridge tape from the SPU, the data files may be read directly by the script. If using cartridge tape from the system, 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 system. 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/rct0e*. When writing from the 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_build*, 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 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 / Module Rev: 1.23, Version: 1 / CPU SN 17
mode-of-operation = normal-os          boot-device = disk
location-of-bootstrap = default      power-up-reboot = enable
automatic-reboot = enable              spu-selftest = enable
os-flags = 0                      remote-port-bps = 1200
(fp)> boot
SPU UNIX boot
: <RETURN>
```

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 UNIX 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 prompt with a ":" on the system console for the device/file specification of the UNIX 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 UNIX 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*, *1-copy*, *2-copy*, and *3-copy*, respectively. The copy used for the primary boot can be selected by setting the *location-of-bootstrap* switch to *default* (disk or tape), *1-copy* (disk or tape), *2-copy* (tape only), or *3-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)**

NAME

update – periodically update the super block

SYNOPSIS

**/etc/update**

DESCRIPTION

**Update** is a program that executes the **sync(2)** primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

SEE ALSO

**sync(2)**
**sync(8)**
**init(8)**

# Appendix A
# Reporting Problems

## A.1 Overview

This appendix introduces the CONVEX Technical Assistance Center (TAC) and the *contact* utility. The *contact* utility is an online system for reporting problems to the TAC. To learn *contact* by using it, enter **contact** at the system prompt and then answer the questions as they appear on the screen. To find out more about using *contact,* read through this appendix. It describes prerequisites and tips for using *contact* and the step-by-step process *contact* takes you through.

## A.2 Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address the diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation problem, contact the TAC. This group stands ready to solve such problems.

## A.3 The *contact* Utility

The TAC recommends using the *contact* utility to report a hardware, software, or documentation problem. The *contact* utility is an interactive utility that helps the TAC track reports and route them to the the CONVEX personnel most qualified to fix them.

After invoking *contact,* it prompts for information about the problem. When you finish your report, *contact* electronically mails it to the TAC. You are notified within 48 hours that the TAC has received your report.

## A.4 Prerequisites

To use *contact* requires

- a UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- the full path name of the program or utility in question
- the version number of the program or utility in question

### A.4.1 UUCP Connection

Before using *contact,* check with your system administrator to be sure there is a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX system to another. The *uucp* (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

### A.4.2 Finding the Program Path Name

To determine the full path name of the program or utility in question, use the *which* command. The following screen illustrates using the *which* command to find the full path name of the loader (*ld*) utility:

```
>which ld
/bin/ld
>
```

In this example, the full path name of the loader is */bin/ld*.

For more information on the *which* command, refer to the *which*(1) man page. You can also use the *info* online information system. Enter **info which** at the system prompt. If you use the C shell (*csh*), you can also use the *whence* command to find the program path name. The *whence* command works like *which*, only faster.

### A.4.3 Finding the Program Version Number

To determine the version number of the program or utility in question, use the *vers* command. The following screen illustrates using the *vers* command (enter **vers**, then the path name of the program or utility) to find the version number of the loader (*ld*) utility.

```
>vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader utility version number is 7.0.

For more information on the *vers* command, refer to the *vers*(1) man page. You can also use the *info* online information system. To do so, enter **info vers** at the system prompt.

## A.5 Tips on Using the *contact* Utility

The *contact* utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to

- use a *.contact* file
- abort a contact session
- resubmit an aborted report
- suspend a contact session
- move from one prompt to another
- use tilde-escape sequences in the *contact* utility

### A.5.1 Using a *.contact* File

When invoked, *contact* prompts for information regarding the problem. The first prompt is for your name, title, phone number, and company name. You can, however, create a *.contact* file to skip this first prompt. Follow these steps:

1. Create a *.contact* file in your home directory.
2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke *contact*, it automatically includes the *.contact* file as input for the first prompt and proceeds to the next prompt.

### A.5.2 Aborting the Report

To abort a contact report, either enter the interrupt key (usually CTRL-C) or choose the abort option when prompted by the *contact* utility. Using CTRL-C to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named *dead.report* in your home directory.

### A.5.3 Submitting the *dead.report* File

When aborting a contact session, the *contact* utility saves the report in a file named *dead.report* in your home directory. Using the *contact* command with the *-r* option automatically merges the contents of the *dead.report* file into the new contact session. Enter

**contact -r**

and *contact* finds the *dead.report* file in your home directory and merges it into the contact report. You can then edit the report. When you end the editing session, *contact* returns to the final prompt, which asks you to review, edit, submit, or abort the report.

### A.5.4 Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press CTRL-Z. To return to the contact session, enter **fg**. Using CTRL-Z and the *fg* (foreground) command lets you switch back and forth between the *contact* utility and the shell. You cannot, however, use CTRL-Z and *fg* to switch back and forth if you are using a Bourne shell (*sh*).

### A.5.5 Ending a Response

The *contact* utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press RETURN. Other prompts require more than a one-line response; to move to the next prompt, press CTRL-D.

### A.5.6 Tilde-Escape Sequences

The *contact* utility treats input beginning with a tilde ( ~ ) as a special sequence. The character following the tilde is considered a request for a special function. The following tilde sequences are recognized by *contact*:

| | |
|---|---|
| **~e** | Start the text editor (defined in your EDITOR environment variable). |
| **~h** | Display a list of available tilde-escape sequences. |
| **~p** | Print the contact report to the terminal screen. |
| **~r** *filename* | Read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence only works for prompts that allow more a than one-line response. |
| **~~** | Insert a single tilde as the first character in the line. |

## A.6 Using the *contact* Utility

The *contact* utility prompts for the following information:

- your name, title, phone number, and corporate name
- the name and version of the product involved
- a one-line summary of the problem
- a detailed description of the problem
- the priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation supporting the problem
- files to include in the contact report

The following is a step-by-step discussion of these prompts:

1a. To invoke the *contact* utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software, or documentation question. The following screen illustrates the *contact* command and the system response:

```
>contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

1b. If there is a *.contact* file in your home directory, *contact* skips the first prompt. The following screen illustrates the *contact* command and the system response when a *.contact* file is in your home directory:

```
>contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

2. The *contact* utility prompts for the version number of the product. If you do not know the version number, use CTRL-Z to suspend the session. Use the *which* (or *whence* if using *csh*) and *vers* commands to find the version number of the product. Use the *fg* command to return to the session and enter the version number in the form X.X or X.X.X.X.
3. The *contact* utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Make this summary as descriptive as possible in one line.
4. The *contact* utility prompts for a detailed description of the problem. Make this description as complete as possible. Include source code and a stack backtrace whenever possible. (Refer to the *adb*(1) or *csd*(1) man page for information on obtaining a stack backtrace.) The more information provided, the quicker the TAC can isolate and solve the problem.
5. The *contact* utility prompts for the priority of the problem. The following screen illustrates this prompt and the priority levels from which to choose; you must enter a priority number.

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

6. The *contact* utility prompts for an explanation of how to reproduce the problem. Include the command syntax and options you used and anything else you did to make your program run.
7. The *contact* utility prompts for any other pertinent comments. Include any relevant information.
8. The *contact* utility prompts for suggestions regarding the documentation supporting the product. Indicate if the documentation could be revised to address the question.
9. The *contact* utility asks for the names of files necessary to reproduce the problem. The following screen illustrates the *contact* prompt and sample user response:

```
Are there any files that should be included in this report (yes | no)?
>yes
Please enter the names of the files, one to a line (^D to terminate)
>test.f
>~/subroutines/sub.f
>
```

> **NOTE**
>
> Tilde-escape sequences are not recognized in responses to this prompt. Instead, *contact* treats a tilde in this section to mean your home directory. This convention is based on use of the tilde for expanding file names in *csh*.

If the files specified are small text files, they are automatically included in the contact report. If the files are too big to be included in this report, *contact* gives further instructions on how to submit these files.

To specify a directory, combine the directory files into a single file using the *tar* command (refer to the *tar*(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

10. The *contact* utility prompts you to review, edit, submit, or abort the contact report. The following screen illustrates this prompt:

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review — Review the text of your contact report. You are then prompted again to select an option.

Edit — Edit the text of the contact report. If you choose to edit the report, *contact* puts you in your default text editor.

Submit — Send the report to the CONVEX TAC. You are notified within 48 hours that the TAC has received the report. This option exits the *contact* utility and returns you to the shell environment.

Abort — Save the text of your report in a file named *dead.report* in your home directory. This option exits the *contact* utility and returns you to the shell environment.

# Index

## W

# Electronic Mail

## Using Electronic Mail

The Hardware Documentation Group has an email address for documentation comments. Use this service to give us a quick response mechanism if you have special documentation questions that you would like addressed immediately. If you have a technical question, you should still contact the Technical Assistance Center, as described previously. To use email response service, just send mail addressed to:

```
cnvxhwdoc@convex.COM
```

We will read your comments and give you a personal reply.

## What to Include in an Email Message

When you use the electronic mail service, please provide the following information:

- The reader's name and company name
- A return email address in INTERNET notation or UUCP (bang) notation
- The manual that is being critiqued
- The chapter and page number in question
- The comment

## Reader's Forum

If you wish to mail your comments to us, please use the form on the next page and list the document page number with your questions and comments. Thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

**CONVEX SPU UNIX Utilities Manual**
Document No. 760-000530-000, Second Edition

## Reader's Forum

Please use this form to submit comments or questions concerning the clarity and service of this manual. Constructive critical comments are most welcome and help us continue in our efforts to generate quality customer documentation. Please list the page number for questions or comments.

_____

_____

_____

_____

_____

_____

_____

_____

**From:**

Name_____ Title_____

Company_____ Date_____

Address and Phone No._____

**FOR ADDITIONAL INFORMATION OR DOCUMENTATION:**

| Location | Phone Number |
|---|---|
| From all locations in continental U.S. | 1(800)952-0379 |
| From locations in Alaska, Hawaii, & Canada | 1(214)497-4379 |
| From all other locations | Contact nearest CONVEX office |

Direct mail orders to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

(Fold Here First)


NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

# BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851

(Fold Here Second)

(Tape or Staple)