999-801-312IS
For use with 3.51 Software

AT&T UNIX® PC

UNIX System V User's Manual Volume I

NOTICE

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

PREFACE

The *AT&T UNIX System V User's Manual* is a two-volume reference manual that describes the operating system capabilities of the AT&T **UNIX*** PC. It provides the UNIX programmer or operating system user with an overview of this implementation and details of commands, subroutines, and other facilities.

This issue of these manuals document version 3.5 of the UNIX PC software.

The Programmer's Manual describes general purpose UNIX commands and programs. This manual is further subdivided as follows:

Section 1	Commands and Application Programs
Section 2	System Calls
Section 3	Subroutines
Section 4	File Formats
Section 5	Miscellaneous Facilities

The Administrator's Manual describes commands and facilities that are used for administrative maintenance of the UNIX system. This manual is further divided as follows:

Section 1M	System Maintenance Commands
Section 7	Special Files
CURSES	Curses/terminfo Programmer's Guide

How to Use These Manuals

The Table of Contents in each manual lists the commands and other facilities in alphabetical order along with brief definitions. Once you have identified a command by the definition, proceed to that section number in the manual. If you are not familiar with the UNIX system commands and facilities, refer to the Permuted Index.

The Programmer's Manual and the Administrator's Manual each contain a Permuted Index, which is an alphabetical listing of the

---

* UNIX is a registered trademark of AT&T

contents grouped by key words. Locate the topic for which you seek information in the middle column of the index, then look to the left column for amplifying information and to the right column for the section number. Proceed to that section number for a full description of the topic.

Version 3.5 UNIX software passes SVVS for System V Release 2. The differences between Version 3.5 for the UNIX PC and System V Release 2 are summarized below.

Section 1M:

acct(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
acctcms(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
acctcon(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
acctmerg(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
acctprc(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
acctsh(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
bdblk(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
brc(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

ckeckall(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
cpset(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
crash(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
dcopy(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
diskusg(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
dismount(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
errdead(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
errdemon(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
errpt(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
errstop(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
filesave(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

fuser(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
fwtmp(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
iv(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
install(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
lddrv(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
link(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
masterupd(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
mkboot(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
mvdir(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
nscloop(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
nscmon(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

profiler(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
pwck(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
qasurvey(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
rboot(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
rc(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5. The command rc(1M) is a subset of brc(1M).
runacct(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sadp(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sar(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sfont(1M)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
st(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
stgetty(1M)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

sysdefs(1M) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

tic(1M) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

uucico(1M) This command is not documented (but is available) on System V Release 2, and is available on the UNIX PC for Version 3.5.

vpmsave(1M) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

vpmset(1M) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

x25pvc(1M) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

Section 7:

acu(7) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

drivers(7) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

escape(7) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

kbd(7) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

ktune(7) This command is not available on System V Release 2, but is available on the UNIX PC

	for Version 3.5.
nc(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
nsc(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
phone(7)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
phonedvr(7)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
prf(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
st(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
stermio(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sxt(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
trace(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
vpm(7)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
window(7)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

x25(7) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

Section 8:

mk(8) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

rje(8) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

Section 1:

acctom(1) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

at(1) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

bs(1) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

calendar(1) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

cat(1) The -v, -t, and -e options are not available on the UNIX PC Version 3.5.

cc(1) The -T, -G, and -# options are not available in System V Release 2.

cfont(1) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

clear(1) This command is not available on System V Release 2, but is available on the UNIX PC

	for Version 3.5.
cpio(1)	The K, R, O, J, and x options are not available in System V Release 2.
ct(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
ctrace(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
cu(1)	The -n option is not available on the UNIX PC Version 3.5.
diff(1)	The -l, -r, -s,-D, and -c options are not available on the UNIX PC Version 3.5.
dircmp(1)	The -wn option is not available on the UNIX PC Version 3.5.
dump(1)	The -g, -c, -p, and -u options are not available on the UNIX PC Version 3.5.
ed(1)	The -p string option is not available on the UNIX PC Version 3.5.
efl(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
eqn(1)	The -T option is not available on the UNIX PC Version 3.5.
f77(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
fc(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
find(1)	The -inum option is not available on the UNIX PC Version 3.5.

fsplit(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
gdev(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
ged(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
get(1)	The –w option is not available on the UNIX PC Version 3.5.
graph(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
graphics(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
greek(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
grep(1)	The –i, –e, and –f options are not available on the UNIX PC Version 3.5.
gutil(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
head(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
hpio(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
ksh(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

ld(1)	The -z, -Z, -T, and -F options are not available in System V Release 2.
lint(1)	The -c and -o options are not available on the UNIX PC Version 3.5.
ls(1)	The -o and -p options are not available on the UNIX PC Version 3.5.
machid(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
mailx(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
message(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
more(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
news(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
nscstat(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
nsctorje(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
nusend(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
path(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

pg(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
prs(1)	The c option is not available on the UNIX PC Version 3.5.
ratfor(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
rjestat(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sag(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sar(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
scrset(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
send(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sh(1)	The a, f, and h options are not available on the UNIX PC Version 3.5.
shform(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
sno(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sort(1)	The –y, –z, and –M options are not available on the UNIX PC Version 3.5.

spell(1)	The –i option is not available on the UNIX PC Version 3.5.
stat(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
stlogin(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
ststat(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
timex(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
toc(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
tplot(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
tput(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
trenter(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
troff(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
tset(1)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
uahelp(1)	This command is not available on System V Release 2, but is available on the UNIX PC

	for Version 3.5.
osend(1)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
vi(1)	The vedit option is not available on the UNIX PC Version 3.5.
who(1)	The –H and –g options are not available on the UNIX PC Version 3.5.

Section 2:

locking(2)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
syslocal(2)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

Section 3:

abs(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2. Note that is a FORTRAN library; most functions are available in the C library.
acos(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
aimag(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
aint(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
asin(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on

	System V Release 2.
atan(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
atan2(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
atof(3c)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
bool(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
conjg(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
cos(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
cosh(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
dim(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
dprod(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
eprintf(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
form(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

ftape(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
getarg(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
getpent(3f)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
iargc(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
index(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
ldgetname(3x)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
len(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
lockf(3c)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
log(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
log10(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
max(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

mclock(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
min(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
menu(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
message(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
mod(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
paste(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
plot(3x)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
rand(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sign(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
signac(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sin(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

sinh(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
sqrt(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
stdio(3s)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
tam(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
tan(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
tanh(3f)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
track(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
wind(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
wrastop(3t)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
x25alnk(3c)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
x25clnk(3c)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

x25hlnk(3c) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

x25ipvc(3c) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

Section 4:

acct(4) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

adf(4) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

errfile(4) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

font(4) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

gps(4) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

piot(4) This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

phone(4) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

shlib(4) This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

term(4) This command is not available on the UNIX PC for Version 3.5, but is available on

	System V Release 2.
terminfo(4)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
ua(4)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.

Section 5:

math(5)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
modemcap(5)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
mptx(5)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
prof(5)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.
termcap(5)	This command is not available on System V Release 2, but is available on the UNIX PC for Version 3.5.
values(5)	This command is not available on the UNIX PC for Version 3.5, but is available on System V Release 2.

TABLE OF CONTENTS

1. Commands and Application Programs

2. System Calls

3. Subroutines

4. File Formats

5. Miscellaneous Facilities

NAME

intro – introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, publicly-accessible commands. Certain distinctions of purpose are made in the headings:

(1) Commands of general utility.
(1C) Commands for communication with other systems.
(1G) Commands used primarily for graphics and computer-aided design.

COMMAND SYNTAX

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

name [*option*(*s*)] [*cmdarg*(*s*)]
where:

name	The name of an executable file.
option	– *noargletter*(*s*) or, – *argletter*<>*optarg* where <> is optional white space.
noargletter	A single letter representing an option without an argument.
argletter	A single letter representing an option requiring an argument.
optarg	Argument (character string) satisfying preceding *argletter*.
cmdarg	Path name (or other command argument) *not* beginning with – or, – by itself indicating the standard input.

SEE ALSO

getopt(1), getopt(3C).

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and 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 and non-zero 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.

BUGS

Regretfully, many commands do not adhere to the aforementioned syntax.

NAME

300, 300s – handle special functions of DASI 300 and 300s terminals

SYNOPSIS

300 [**+12**] [**−n**] [**−d***t*,*l*,*c*]

300s [**+12**] [**−n**] [**−d***t*,*l*,*c*]

DESCRIPTION

300 supports special functions and optimizes the use of the DASI 300 (GSI 300 or DTC 300) terminal; *300s* performs the same functions for the DASI 300s (GSI 300s or DTC 300s) terminal. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. It also attempts to draw Greek letters and other special symbols. It permits convenient use of 12-pitch text. It also reduces printing time 5 to 70%. *300* can be used to print equations neatly, in the sequence:

neqn file . . . | nroff | 300

WARNING: if your terminal has a PLOT switch, make sure it is turned *on* before *300* is used.

The behavior of *300* can be modified by the optional flag arguments to handle 12-pitch text, fractional line spacings, messages, and delays.

+12 permits use of 12-pitch, 6 lines/inch text. DASI 300 terminals normally allow only two combinations: 10-pitch, 6 lines/inch, or 12-pitch, 8 lines/inch. To obtain the 12-pitch, 6 lines per inch combination, the user should turn the PITCH switch to 12, and use the **+12** option.

−*n* controls the size of half-line spacing. A half-line is, by default, equal to 4 vertical plot increments. Because each increment equals 1/48 of an inch, a 10-pitch line-feed requires 8 increments, while a 12-pitch line-feed needs only 6. The first digit of *n* overrides the default value, thus allowing for individual taste in the appearance of subscripts and superscripts. For example, *nroff* half-lines could be made to act as quarter-lines by using **−2**. The user could also obtain appropriate half-lines for 12-pitch, 8 lines/inch mode by using the option **−3** alone, having set the PITCH switch to 12-pitch.

−d*t*,*l*,*c* controls delay factors. The default setting is **−d3,90,30**. DASI 300 terminals sometimes produce peculiar output when faced with very long lines, too many tab characters, or long strings of blankless, non-identical characters. One null (delay) character is inserted in a line for every set of *t* tabs, and for every contiguous string of *c* non-blank, non-tab characters. If a line is longer than *l* bytes, 1+(total length)/20 nulls are inserted at the end of that line. Items can be omitted from the end of the list, implying use of the default values. Also, a value of zero for *t* (*c*) results in two null bytes per tab (character). The former may be needed

for C programs, the latter for files like **/etc/passwd**. Because terminal behavior varies according to the specific characters printed and the load on a system, the user may have to experiment with these values to get correct output. The **−d** option exists only as a last resort for those few cases that do not otherwise print properly. For example, the file **/etc/passwd** may be printed using **−d3,30,5**. The value **−d0,1** is a good one to use for C programs that have many levels of indentation.

Note that the delay control interacts heavily with the prevailing carriage return and line-feed delays. The *stty*(1) modes **nl0 cr2** or **nl0 cr3** are recommended for most uses.

300 can be used with the *nroff* **−s** flag or **.rd** requests, when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the return key in these cases, you must use the line-feed key to get any response.

In many (but not all) cases, the following sequences are equivalent:

nroff −T300 files . . . and nroff files . . . | 300
nroff −T300−12 files . . . and nroff files . . . | 300 +12

The use of *300* can thus often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of *300* may produce better-aligned output.

The *neqn* names of, and resulting output for, the Greek and special characters supported by *300* are shown in *greek*(5).

SEE ALSO

450(1), eqn(1), mesg(1), nroff(1), stty(1), tabs(1), tbl(1), greek(5).

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.
If your output contains Greek and/or reverse line-feeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters and misaligning the first line of text after one or more reverse line-feeds.

NAME

4014 – paginator for the Tektronix 4014 terminal

SYNOPSIS

4014 [**−t**] [**−n**] [**−cN**] [**−pL**] [file]

DESCRIPTION

The output of *4014* is intended for a Tektronix 4014 terminal; *4014* arranges for 66 lines to fit on the screen, divides the screen into *N* columns, and contributes an eight-space page offset in the (default) single-column case. Tabs, spaces, and backspaces are collected and plotted when necessary. TELETYPE Model 37 half- and reverse-line sequences are interpreted and plotted. At the end of each page, *4014* waits for a new-line (empty line) from the keyboard before continuing on to the next page. In this wait state, the command **!***cmd* will send the *cmd* to the shell.

The command line options are:

−t Don't wait between pages (useful for directing output into a file).

−n Start printing at the current cursor position and never erase the screen.

−c*N* Divide the screen into *N* columns and wait after the last column.

−p*L* Set page length to *L*; *L* accepts the scale factors **i** (inches) and **l** (lines); default is lines.

SEE ALSO

pr(1), tc(1).

NAME

450 – handle special functions of the DASI 450 terminal

SYNOPSIS

450

DESCRIPTION

450 supports special functions of, and optimizes the use of, the DASI 450 terminal, or any terminal that is functionally identical, such as the DIABLO 1620 or XEROX 1700. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. It also attempts to draw Greek letters and other special symbols in the same manner as *300*(1). *450* can be used to print equations neatly, in the sequence:

neqn file . . . | nroff | 450

WARNING: make sure that the PLOT switch on your terminal is ON before *450* is used. The SPACING switch should be put in the desired position (either 10- or 12-pitch). In either case, vertical spacing is 6 lines/inch, unless dynamically changed to 8 lines per inch by an appropriate escape sequence.

450 can be used with the *nroff* **–s** flag or **.rd** requests, when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the return key in these cases, you must use the line-feed key to get any response.

In many (but not all) cases, the use of *450* can be eliminated in favor of one of the following:

nroff –T450 files . . .

or

nroff –T450–12 files . . .

The use of *450* can thus often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of *450* may produce better-aligned output.

The *neqn* names of, and resulting output for, the Greek and special characters supported by *450* are shown in *greek*(5).

SEE ALSO

300(1), eqn(1), mesg(1), nroff(1), stty(1), tabs(1), tbl(1), greek(5).

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.
If your output contains Greek and/or reverse line-feeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters and misaligning the first line of text after one or more reverse line-feeds.

NAME

adb – absolute 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 **a.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 Hexadecimal by default or if preceded by 0x; octal if preceded by 0o or 0O; decimal if preceded by 0t or 0T.

integer.fraction
A 32 bit floating point number.

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

< *name*
The value of *name*, which is either a variable name or a 68010/68020 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 registers are **d0** through **d7**, **a0** through **a7**, **sp**, **pc**, **cc**,

sr, and **usp**.

symbol A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. \ may be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objfil*.

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

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

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

Monadic operators:

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

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

-exp Integer negation.

~exp Bitwise complement.

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

e1+e2 Integer addition.

e1-e2 Integer subtraction.

*e1*e2* Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

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

COMMANDS

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

?f Locations starting at *address* in *objfil* are printed according to the format *f*. *dot* is incremented by the sum of the increments for each format letter (q.v.).

/f Locations starting at *address* in *corfil* are printed according to the format *f* and *dot* is incremented as for ?.

=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 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.
f 4 Print the 32 bit value as a floating point number.
F 8 Print double floating point.
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. The value *n* is the length of the string including its zero terminator.
Y 4 Print 4 bytes in date format (see *ctime*(3C)).
i n Print as machine instructions. The value *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 new-line.

"..." 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.

new-line

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 **fp**). 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).

o All integers input are regarded as octal.
d Reset integer input as described in *EXPRESSIONS.*
q Exit from *adb*.
v Print all non zero variables.
f Print the 68881 floating-point registers.
m Print the address map.

:*modifier*

Manage a subprocess. Available modifiers are:

b*c* 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. The value *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* (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 (0407, 0410 or 0413).
s	The stack segment size.
t	The text segment size.

ADDRESSES

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

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

otherwise

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

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

The initial setting of both mappings is suitable for normal **a.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.

In order for *adb* to be used on large files all appropriate values are kept as signed 32-bit integers.

FILES

/dev/kmem
/dev/swap
a.out
core

SEE ALSO

ptrace(2), a.out(4), core(4).

DIAGNOSTICS

"Adb" when there is no current command or format. Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero 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

admin – create and administer SCCS files

SYNOPSIS

admin [**–n**] [**–i**[name]] [**–r**rel] [**–t**[name]] [**–f**flag[flag-val]] [**–d**flag[flag-val]] [**–a**login] [**–e**login] [**–m**[mrlist]] [**–y**[comment]] [**–h**] [**–z**] files

DESCRIPTION

Admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters **s.**). If a named file doesn't exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

–n This keyletter indicates that a new SCCS file is to be created.

–i[*name*] The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see **–r** keyletter for delta numbering scheme). If the **i** keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an *admin* command on which the **i** keyletter is supplied. Using a single *admin* to create two or more SCCS files require that they be created empty (no **–i** keyletter). Note that the **–i** keyletter implies the **–n** keyletter.

–r*rel* The *rel*ease into which the initial delta is inserted. This keyletter may be used only if the **–i** keyletter is also used. If the **–r** keyletter is not used, the initial delta is inserted into release 1. The level of the ini-

tial delta is always 1 (by default initial deltas are named 1.1).

-t[*name*] The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **-t** keyletter is used and *admin* is creating a new SCCS file (the **-n** and/or **-i** keyletters also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **-t** keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **-t** keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

-f*flag* This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **f** keyletters may be supplied on a single *admin* command line. The allowable *flag*s and their values are:

b Allows use of the **-b** keyletter on a *get*(1) command to create branch deltas.

c*ceil* The highest release (i.e., "ceiling"), a number less than or equal to 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **c** flag is 9999.

f*floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **f** flag is 1.

d*SID* The default delta number (SID) to be used by a *get*(1) command.

i Causes the "No id keywords (ge6)" message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(1)) are found in the text retrieved or stored in the SCCS file.

j Allows concurrent *get*(1) commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to which deltas can no longer be made (**get -e** against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= *RELEASE NUMBER* | **a**

The character **a** in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file preventing branch deltas from being created from them in the future.

q*text* User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*(1).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*(1). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* *Type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*(1).

v[*pgm*] Causes *delta*(1) to prompt for Modification Request (*MR*) numbers as the reason for creating a delta. The optional value specifies the name of an *MR* number validity checking program (see *delta*(1)). (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is null).

−d*flag* Causes removal (deletion) of the specified *flag* from an SCCS file. The **−d** keyletter may be specified only when processing existing SCCS files. Several **−d** keyletters may be supplied on a single *admin* command. See the **−f** keyletter for allowable *flag* names.

l*list* A *list* of releases to be "unlocked". See the **−f** keyletter for a description of the **l** flag and the syntax of a *list*.

−a*login* A *login* name, or numerical UNIX group ID, to be added to the list of users which may

make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas.

−e*login* A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **e** keyletters may be used on a single *admin* command line.

−y[*comment*] The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(1). Omission of the **−y** keyletter results in a default comment line being inserted in the form:

date and time created *YY/MM/DD HH:MM:SS* by *login*

The **−y** keyletter is valid only if the **−i** and/or **−n** keyletters are specified (i.e., a new SCCS file is being created).

−m[*mrlist*] The list of Modification Requests (*MR*) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(1). The **v** flag must be set and the *MR* numbers are validated if the **v** flag has a value (the name of an *MR* number validation program). Diagnostics will occur if the **v** flag is not set or *MR* validation fails.

−h Causes *admin* to check the structure of the SCCS file (see *sccsfile*(5)), and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

−z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **−h**, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

FILES

The last component of all SCCS file names must be of the form **s.***file-name*. New SCCS files are given mode 444 (see *chmod*(1)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called **x.***file-name*, (see *get*(1)), created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of *ed*(1). *Care must be taken!* The edited file should *always* be processed by an **admin −h** to check for corruption followed by an **admin −z** to generate a proper check-sum. Another **admin −h** is recommended to ensure the SCCS file is valid.

Admin also makes use of a transient lock file (called **z.***file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ar – archive and library maintainer for portable archives

SYNOPSIS

ar key [posname] afile name ...

DESCRIPTION

The *Ar* command maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by *ar* consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

When *ar* creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure is described in detail in *ar*(4). The archive symbol table (described in *ar*(4)) is used by the link editor (*ld*(1)) to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by *ar* when there is at least one object file in the archive. The archive symbol table is in a specifically named file which is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever the *ar*(1) command is used to create or update the contents of such an archive, the symbol table is rebuilt. The **s** option described below will force the symbol table to be rebuilt.

Key is an optional –, followed by one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcls**. *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set **abi** is used, then the *posname* argument must be present and specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, it gives a long listing of all information about the files. When used with **x**, precede each file with a name.

c Suppress the message that is produced by default when *afile* is created.

l Place temporary files in the local current working directory rather than in the directory specified by the environment variable TMPDIR or in the default directory

s Force the regeneration of the archive symbol table even if *ar*(1) is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after the *strip*(1) command has been used on the archive.

FILES

/tmp/ar* temporaries

SEE ALSO

file(1), ld(1), lorder(1), strip(1), a.out(4), ar(4).

NOTES

This archive format is new to this release. The *ar* command will not accept archive files in the old format. Use Release 2.0 or 3.0 *ar* commands (available from AT&T on an installable floppy disk) to take apart archive files in the old format. Release 3.5 utilities cannot be used to modify Release 3.0 archives; Release 3.0 utilities cannot be used to modify Release 3.5 archives.

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

as – assembler

SYNOPSIS

as [**–o** objfile] [**–n**] [**–j**] [**–m**] [**–R**] [**–r**] [**–bwl**] [**–V**] [**–T**] sourcefile

DESCRIPTION

The *as* command translates mc68010 or mc68020 assembly language in *sourcefile* into object code. The result is a common object file, suitable for input to the link editor. The following flags may be specified in any order:

–o *objfile*
: Put the output of the assembly in *objfile*. By default, the output file name is formed by removing the **.s** suffix, if there is one, from the input file name and appending a **.o** suffix.

–n
: Turn off long/short address optimization. By default, address optimization takes place.

–j
: Invoke the long-jump assembler. The address optimization algorithm chooses between long and short address lengths, with short lengths chosen when possible. Often, three distinct lengths are allowed by the machine architecture; a choice must be made between two of those lengths. When the two choices given to the assembler exclude the largest length allowed, then some addresses might be unrepresentable. The long-jump assembler will always have the largest length as one of its allowable choices. If the assembler is invoked without this option, and the case arises where an address is unrepresentable by either of the two allowed choices, then the user will be informed of the error, and advised to try again using the **–j** option.

–m
: Run the *m4* macro pre-processor on the input to the assembler.

–R
: Remove (unlink) the input file after assembly is completed.

–r
: Place all assembled data (normally placed in the **data** section) into the **text** section. This option effectively disables the **.data** pseudo operation. This option is off by default.

–[bwl]
: Create byte **(b)**, halfword **(w)** or long **(l)** displacements for undefined symbols. (An undefined symbol is a reference to a symbol whose definition is external to the input file or a forward reference.) The default value for this option is long **(l)** displacements.

–V
: Write the version number of the assembler being run on the standard error output.

–T
: Truncate symbols to eight characters.

FILES

/usr/tmp/as[1-6]*XXXXXX* temporary files

SEE ALSO

ld(1), m4(1), nm(1), strip(1), a.out(4).

WARNING

If the **-m** (*m4* macro pre-processor invocation) option is used, keywords for *m4* (see *m4*(1)) cannot be used as symbols (variables, functions, labels) in the input file since *m4* cannot determine which are assembler symbols and which are real *m4* macros.

Use the **-b** or **-w** option only when undefined symbols are known to refer to locations representable by the specified default displacement. Use of either option when assembling a file containing a reference to a symbol that is to be resolved by the loader can lead to unpredictable results, since the loader may be unable to place the address of the symbol into the space provided.

BUGS

The **.align** assembler directive is not guaranteed to work in the **.text** section when optimization is performed.

Arithmetic expressions may only have one forward referenced symbol per expression.

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [files]

DESCRIPTION

Asa interprets the output of FORTRAN programs that utilize ASA carriage control characters. It processes either the *files* whose names are given as arguments or the standard input if no file names are supplied. The first character of each line is assumed to be a control character; their meanings are:

' '	(blank) single new line before printing
0	double new line before printing
1	new page before printing
+	overprint previous line.

Lines beginning with other than the above characters are treated as if they began with **' '**. The first character of a line is *not* printed. If any such lines appear, an appropriate diagnostic will appear on standard error. This program forces the first line of each input file to start on a new page.

To correctly view the output of FORTRAN programs which use ASA carriage control characters, *asa* could be used as a filter thusly:

```
a.out | asa | lpr
```

and the output, properly formatted and pagenated, would be directed to the line printer. FORTRAN output sent to a file could be viewed by:

```
asa file
```

SEE ALSO

efl(1).

NAME

async_main – vt100, b513 terminal emulation program

SYNOPSIS

async_main e *profile*
async_main r *profile* **f /dev/tty000**
async_main r *profile* **f /dev/ph**x **i** *phfd* **p** *pid*

DESCRIPTION

Async_main is used to communicate with a host computer using vt100 or b513 escape and control sequences and to edit communication profiles. *Async_main* may be used with any of ports **/dev/tty000** or **/dev/ph**x, where x is 0 or 1, to communicate with a host.

Use the first form above to edit (**e**) a profile. *Profile* must contain the full path name of the file to be edited. The profile suffix for **/dev/tty000** is **:A2** and for **/dev/ph**x the suffix is **:Am**.

To run *async_main* via **/dev/tty000**, use the second form above. Again, *profile* must contain the full path name of the file to be used in the run (**r**) session.

To run *async_main* via **/dev/ph0** or **/dev/ph1**, use the third form above. *Phfd* is the phone file descriptor. The calling process must first open the device file via *open*(2) and pass the phone file descriptor to *async_main*. *Pid* is the process ID of the calling process.

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [**–Fc**] [prog] [parameters] [files]

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as **–f** *file*. The *prog* string should be enclosed in single quotes (') to protect it from the shell.

Parameters, in the form x=... y=... etc., may be passed to *awk*.

Files are read in order; if there are no files, the standard input is read. The file name – means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, see below). The fields are denoted **$1**, **$2**, . . . ; **$0** refers to the entire line.

A pattern-action statement has the form:

pattern { action }

A missing action means print the line; a missing pattern always matches. An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] . . . }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next    # skip remaining patterns on this input line
exit    # skip the rest of the input
```

Statements are terminated by semicolons, new-lines, or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The C operators ++, --, +=, -=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The *print* statement prints its arguments on the standard output (or on a file if *>expr* is present), separated by the current output

field separator, and terminated by the output record separator. The *printf* statement formats its expression list according to the format (see *printf*(3S)).

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer; *substr*(*s*, *m*, *n*) returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf*(*fmt*, *expr*, *expr*, ...) formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in *egrep* (see *grep*(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions. A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression

where a relop is any of the six relational operators in C, and a matchop is either ~ (for *contains*) or !~ (for *does not contain*). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields by starting the program with:

BEGIN { FS = *c* }

or by using the **-F***c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default new-line); and OFMT, the output format for numbers (default **%.6g**).

EXAMPLES

Print lines longer than 72 characters:

length > 72

Print first two fields in opposite order:

{ print $2, $1 }

Add up first column, print sum and average:

```
        { s += $1 }
END     { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
       { print }
```

command line: awk -f program n=5 input

SEE ALSO

grep(1), lex(1), sed(1).
Awk-A Pattern Scanning and Processing Language by A. V. Aho, B. W. Kernighan, and P. J. Weinberger.

BUGS

Input white space is not preserved on output if fields are involved. There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string (`""`) to it.

NAME

banner – make posters

SYNOPSIS

banner strings

DESCRIPTION

Banner prints its arguments (each up to 10 characters long) in large letters on the standard output.

SEE ALSO

echo(1).

NAME

basename, dirname – deliver portions of path names

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

Basename deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and prints the result on the standard output. It is normally used inside substitution marks (` `) within shell procedures.

Dirname delivers all but the last level of the path name in *string*.

EXAMPLES

The following example, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named **cat** in the current directory:

```
cc $1
mv a.out `basename $1 .c`
```

The following example will set the shell variable NAME to **/usr/src/cmd**:

```
NAME=`dirname /usr/src/cmd/cat.c`
```

SEE ALSO

sh(1).

BUGS

The *basename* of / is null and is considered an error.

NAME

bc – arbitrary-precision arithmetic language

SYNOPSIS

bc [**–c**] [**–l**] [file ...]

DESCRIPTION

Bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The –l argument stands for the name of an arbitrary precision math library. The syntax for *bc* programs is as follows; L means letter a–z, E means expression, S means statement.

Comments

are enclosed in /* and */.

Names

simple variables: L
array elements: L [E]
The words "ibase", "obase", and "scale"

Other operands

arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
length (E) number of significant decimal digits
scale (E) number of digits right of decimal point
L (E , ... , E)

Operators

+ – * / % ^ (% is remainder; ^ is power)
++ –– (prefix and postfix; apply to names)
== <= >= != < >
= =+ =– =* =/ =% =^

Statements

E
{ S ; ... ; S }
if (E) S
while (E) S
for (E ; E ; E) S
null statement
break
quit

Function definitions

```
define L ( L ,..., L ) {
        auto L, ... , L
        S; ... S
        return ( E )
}
```

Functions in –l math library

s(x) sine
c(x) cosine
e(x) exponential

l(x)	log
a(x)	arctangent
j(n,x)	Bessel function

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or newlines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. "Auto" variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

Bc is actually a preprocessor for *dc*(1), which it invokes automatically, unless the **-c** (compile only) option is present. In this case the *dc* input is sent to the standard output instead.

EXAMPLE

```
scale = 20
define e(x){
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1==1; i++){
                a = a*x
                b = b*i
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

defines a function to compute an approximate value of the exponential function and

```
for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

FILES

/usr/lib/lib.b	mathematical library
/usr/bin/dc	desk calculator proper

SEE ALSO

dc(1).

BC-An Arbitrary Precision Desk-Calculator Language by L. L. Cherry and R. Morris.

BUGS

No &&, | | yet.
For statement must have all three E's.
Quit is interpreted when read, not when executed.

NAME

bdiff – big diff

SYNOPSIS

bdiff file1 file2 [n] [**–s**]

DESCRIPTION

Bdiff is used in a manner analogous to *diff*(1) to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for *diff*. *Bdiff* ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. The value of *n* is 3500 by default. If the optional third argument is given, and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail. If *file1* (*file2*) is –, the standard input is read. The optional **–s** (silent) argument specifies that no diagnostics are to be printed by *bdiff* (note, however, that this does not suppress possible exclamations by *diff*. If both optional arguments are specified, they must appear in the order indicated above.

The output of *bdiff* is exactly that of *diff*, with line numbers adjusted to account for the segmenting of the files (that is, to make it look as if the files had been processed whole). Note that because of the segmenting of the files, *bdiff* does not necessarily find a smallest sufficient set of file differences.

FILES

/tmp/bd?????

SEE ALSO

diff(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

bfs – big file scanner

SYNOPSIS

bfs [–] name

DESCRIPTION

Bfs is (almost) like *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 255 characters per line. *Bfs* is usually more efficient than *ed* for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the **w** command. The optional – suppresses printing of sizes. Input is prompted with * if **P** and a carriage return are typed as in *ed*. Prompting can be turned off again by inputting another **P** and carriage return. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under *ed* are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?: > indicates downward search without wrap-around, and < indicates upward search without wrap-around. Since *bfs* uses a different regular expression-matching routine from *ed*, the regular expressions accepted are slightly wider in scope (see *regcmp*(3X)). There is a slight difference in mark names: only the letters **a** through **z** may be used, and all 26 marks are remembered.

The **e**, **g**, **v**, **k**, **n**, **p**, **q**, **w**, =, **!** and null commands operate as described under *ed*. Commands such as ---, +++-, +++=, **-12**, and **+4p** are accepted. Note that **1,10p** and **1,10** will both print the first ten lines. The **f** command only prints the name of the file being scanned; there is no *remembered* file name. The **w** command is independent of output diversion, truncation, or crunching (see the **xo**, **xt** and **xc** commands, below). The following additional commands are available:

xf *file*

Further commands are taken from the named *file*. When an end-of-file is reached, an interrupt signal is received or an error occurs, reading resumes with the file containing the **xf**. **Xf** commands may be nested to a depth of 10.

xo [*file*]

Further output from the **p** and null commands is diverted to the named *file*, which, if necessary, is created mode 666. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

: *label*

This positions a *label* in a command file. The *label* is terminated by new-line, and blanks between the **:** and the start of the *label* are ignored. This command may also be used to insert comments into a command file, since labels need not be referenced.

(. , .)**xb**/*regular expression*/*label*

A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

1. Either address is not between **1** and $.
2. The second address is less than the first.
3. The regular expression doesn't match at least one line in the specified range, including the first and last lines.

On success, . is set to the line matched and a jump is made to *label*. This command is the only one that doesn't issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. Note that the command

```
xb/^/ label
```

is an unconditional jump.
The **xb** command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe only a downward jump is possible.

xt *number*

Output from the **p** and null commands is truncated to at most *number* characters. The initial number is 255.

xv[*digit*] [*spaces*] [*value*]

The variable name is the specified *digit* following the **xv**. **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. **Xv61,100p** assigns the value **1,100p** to the variable **6**. To reference a variable, put a **%** in front of the variable name. For example, using the above assignments for variables **5** and **6**:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would globally search for the characters **100** and print each line containing a match. To escape the special meaning of **%**, a \ must precede it.

```
g/".*\%[cds]/p
```

could be used to match and list lines containing *printf* of characters, decimal integers, or strings.
Another feature of the **xv** command is that the first line of output from a UNIX command can be stored into a variable. The only requirement is that the first character of *value* be an **!**. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable **5**, print it, and increment the variable **6** by one. To escape the special meaning of **!** as the first character of *value*, precede it with a \.

```
xv7\!date
```

stores the value **!date** into variable **7**.

xbz *label*

xbn *label*

These two commands will test the last saved *return code* from the execution of a UNIX command (*!command*) or nonzero value, respectively, to the specified label. The two examples below both search for the next five lines containing the string **size**.

```
xv55
: l
/size/
xv5!expr %5 - 1
!if 0%5 != 0 exit 2
xbn l
xv45
: l
/size/
xv4!expr %4 - 1
!if 0%4 = 0 exit 2
xbz l
```

xc [*switch*]

If *switch* is **1**, output from the **p** and null commands is crunched; if *switch* is **0** it isn't. Without an argument, **xc** reverses *switch*. Initially *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

SEE ALSO

csplit(1), ed(1), regcmp(3X).

DIAGNOSTICS

? for errors in commands, if prompting is turned off. Self-explanatory error messages when prompting is on.

NAME

cal – print calendar

SYNOPSIS

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.

Beware that "cal 78" refers to the early Christian era, not the 20th century.

NAME

cat – concatenate and print files

SYNOPSIS

cat [**–u**] [**–s**] 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 input file is given, or if the argument – is encountered, *cat* reads from the standard input file. Output is buffered unless the **–u** option is specified. The **–s** option makes *cat* silent about non-existent files. No input file may be the same as the output file unless it is a special file.

WARNING

Command formats such as

cat file1 file2 >file1

will cause the original data in *file1* to be lost, therefore, take care when using shell special characters.

SEE ALSO

cp(1), pr(1).

NAME

cb – C program beautifier

SYNOPSIS

cb [**–s**] [**–j**] [**–l** leng] [file ...]

DESCRIPTION

Cb reads C programs either from its arguments or from the standard input and writes them on the standard output with spacing and indentation that displays the structure of the code. Under default options, *cb* preserves all user new-lines. Under the **–s** flag *cb* canonicalizes the code to the style of Kernighan and Ritchie in *The C Programming Language*. The **–j** flag causes split lines to be put back together. The **–l** flag causes *cb* to split lines that are longer than *leng*.

SEE ALSO

cc(1).

The C Programming Language by B. W. Kernighan and D. M. Ritchie.

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cc - C compiler

SYNOPSIS

cc [option] ... file ...

DESCRIPTION

Cc is the UNIX PC Portable C compiler. It accepts several types of arguments.

Arguments whose names end with **.c** are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with **.o** substituted for **.c**. The **.o** file is normally deleted, however, if a single C program is compiled and loaded all at once.

In the same way, arguments whose names end with **.s** are taken to be assembly source programs and are assembled, producing a **.o** file.

The following options are interpreted by *cc*. See *ld*(1) for link editor options and *cpp*(1) for more preprocessor options.

−# Display without executing each command that *cc* generates.

−c Suppress the link edit phase of the compilation, and force an object file to be produced even if only one program is compiled.

−E Run only *cpp*(1) on the named C programs, and send the result to the standard output.

−g Cause the compiler to generate additional information needed for the use of *sdb*(1).

−o*outfile*
Produce an output object file named *outfile*. The name of the default file is **a.out**. This is a link editor option.

−O Invoke an object-code optimizer.

−p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if link editing takes place, replace the standard startoff routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

−P Run only *cpp*(1) on the named C programs, and leave the result on corresponding files suffixed **.i**.

−S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed **.s**.

−Wc,arg1[,arg2...]
Hand off the argument[s] *argi* to pass *c* where *c* is one of **[p02al]** indicating the preprocessor, complier, optimizer, assembler, or link editor, respectively. For example,

-Wa,-m passes **-m** to the assembler.

The C language standard was extended to include arbitrary length variable names. The **-T** option "**-Wp,-T -W0,-XT**" will cause the current compiler to behave the same as previous compilers with respect to the length of variable names.

-68010
Generate code for the mc68010 processor.

-68000
Generate code for the mc68000 processor.

-v Verbose. Print pass names as they are performed.

-T Truncate variable names to eight characters.

-w Tell the linker (*ld*) not to print warnings about symbols that partially matched.

The C compiler uses one of three code generators for the

CPU=xxxxx,FPU=yyyyy

where CPU indicates the central processor to generate for and FPU indicates the style of floating-point math to use. *xxxxx* must currently be 68010, and *yyyyy* may be 68881 or SOFTWARE. The FPU parameter may be deleted, the default is SOFTWARE. The CENVIRON variable should always be set to the appropriate values in the **.profile** or **.Kshrc** files.

Other arguments are taken to be either link editor option arguments, C preprocessor option arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name **a.out**.

Note that modules appear to *ld* in the same order they (or their source code version) appear to *cc*. Thus a library or object file should appear in the *cc* argument list after any module that refers to it.

FILES

file.c	input file
file.o	object file
file.s	assembly language file
a.out	linked output
/tmp/ctm*	temporary
/lib/cpp	C preprocessor *cpp*(1)
/lib/ccom	compiler
/lib/optim	optional optimizer
/bin/as	assembler, *as*(1)
/bin/ld	link editor, *ld*(1)
/lib/crt0.o	runtime startoff
/lib/crts.o	shared library startoff
/lib/mcrt0.o	profiling startoff
/lib/libc.a	standard C library, see section 3
/lib/libp/lib/*.a	profiled versions of libraries

/lib/crts.o	shared library startoff
/lib/mcrt0.o	profiling startoff
/lib/libc.a	standard C library, see section 3
/lib/libp/lib/*.a	profiled versions of libraries

SEE ALSO

The C Programming Language by B. W. Kernighan and D. M. Ritchie.
adb(1), cpp(1), as(1), ld(1), prof(1), monitor(3C), shlib(4).

NOTES

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value are to explicitly call *exit*(2) or to leave the function **main**() with a "**return** *expression;*" construct.

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or the link editor.

BUGS

The optimizer may produce incorrect code if one *asm()* routine requires a jump to a label in another *asm()* routine. The optimizer should be turned off for these code segments.

NAME

cd – change working directory

SYNOPSIS

cd [directory]

DESCRIPTION

If *directory* is not specified, the value of shell parameter **$HOME** is used as the new working directory. If *directory* specifies a complete path starting with /, ., .., *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the **$CDPATH** shell variable. **$CDPATH** has the same syntax as, and similar semantics to, the **$PATH** shell variable. *Cd* 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; therefore, it is recognized and internal to the shell.

SEE ALSO

pwd(1), sh(1), chdir(2).

NAME

cdc – change the delta commentary of an SCCS delta

SYNOPSIS

cdc **–r**SID [**–m**[mrlist]] [**–y**[comment]] files

DESCRIPTION

Cdc changes the *delta commentary*, for the *SID* specified by the **–r** keyletter, of each named SCCS file.

Delta commentary is defined to be the Modification Request (**MR**) and comment information normally specified via the *delta*(1) command (**–m** and **–y** keyletters).

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of *keyletter* arguments, and file names.

All the described *keyletter* arguments apply independently to each named file:

–r*SID* Used to specify the *S*CCS *ID*entification (*SID*) string of a delta for which the delta commentary is to be changed.

–m[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(1)) then a list of **MR** numbers to be added and/or deleted in the delta commentary of the *SID* specified by the **–r** keyletter *may* be supplied. A null **MR** list has no effect.

MR entries are added to the list of **MR**s in the same manner as that of *delta*(1). In order to delete an **MR**, precede the **MR** number with the character **!** (see *EXAMPLES*). If the **MR** to be deleted is currently in the list of **MR**s, it is removed and changed into a "comment" line. A list of all deleted **MR**s is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If **–m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **–y** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the **MR** list.

Note that if the **v** flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) which validates the correctness of the **MR** numbers. If a non-zero exit status is returned from the **MR** number validation program, *cdc* terminates and the delta commentary remains unchanged.

-y[*comment*] Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the **-r** keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If **-y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the *comment* text.

The exact permissions necessary to modify the SCCS file are documented in the *Source Code Control System User's Guide*. Simply stated, they are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

```
cdc -r1.6 -m"bl78-12345 !bl77-54321 bl79-00001" -ytrouble s.file
```

adds bl78-12345 and bl79-00001 to the **MR** list, removes bl77-54321 from the **MR** list, and adds the comment **trouble** to delta 1.6 of s.file.

```
cdc -r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble
```

does the same thing.

WARNINGS

If SCCS file names are supplied to the *cdc* command via the standard input (- on the command line), then the **-m** and **-y** keyletters must also be used.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

cflow – generate C flow graph

SYNOPSIS

cflow [**–r**] [**–ix**] [**–i_**] [**–d**num] files

DESCRIPTION

Cflow analyzes a collection of C, YACC, LEX, assembler, and object files and attempts to build a graph charting the external references. Files suffixed in **.y**, **.l**, **.c**, and **.i** are YACC'd, LEX'd, and C-preprocessed (bypassed for **.i** files) as appropriate and then run through the first pass of *lint*(1). (The **–I**, **–D**, and **–U** options of the C-preprocessor are also understood.) Files suffixed with **.s** are assembled and information is extracted (as in **.o** files) from the symbol table. The output of all this non-trivial processing is collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference (i.e., line) number, followed by a suitable number of tabs indicating the level. Then the name of the global (normally only a function not defined as an external or beginning with an underscore; see below for the **–i** inclusion option) a colon and its definition. For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char** *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only < > is printed.

As an example, given the following in *file.c*:

```
int     i;

main()
{
        f();
        g();
        f();
}

f()
{
        i = h();
}
```

the command

```
cflow file.c
```

produces the the output

```
1       main: int(), <file.c 4>
2               f: int(), <file.c 11>
3                       h: <>
4                       i: int, <file.c 1>
5               g: <>
```

When the nesting level becomes too deep, the **-e** option of *pr*(1) can be used to compress the tab expansion to something less than every eight spaces.

The following options are interpreted by *cflow*:

-r Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

-ix Include external and static data symbols. The default is to include only functions in the flow graph.

-i_ Include names that begin with an underscore. The default is to exclude these functions (and data if *-ix* is used).

-dnum The *num* decimal integer indicates the depth at which the flow graph is cut off. By default this is a very large number. Attempts to set the cutoff depth to a nonpositive integer will be met with contempt.

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

SEE ALSO

as(1), cc(1), lex(1), lint(1), nm(1), pr(1), yacc(1).

BUGS

Files produced by *lex*(1) and *yacc*(1) cause the reordering of line number declarations which can confuse *cflow*. To get proper results, feed *cflow* the *yacc* or *lex* input.

NAME

cfont – convert fonts to ASCII and vice-versa

SYNOPSIS

cfont binaryfont > asciifont
cfont asciifont > binaryfont

DESCRIPTION

Cfont converts to and from the system font file format (see *font*(4)). When asked to decode a binary font, a line-oriented ASCII file is produced. An example of such a file is included here.

EXAMPLE

```
# magic 0x18e
# flags 0x0
hs   26
vs   38
basel 24

################ Character 0x21, 041, 33   ! ! ! ! ! ! ! ! !
!
char 33
hs  3
vs  19
ha  3
va  -19
hi  8
vi  0
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits ***
bits
bits
bits
bits ***
bits ***
bits ***

################ Character 0x24, 044, 36   $ $ $ $ $ $ $ $ $ $
char 36
hs  13
vs  26
ha  3
```

```
va -23
hi 17
vi 0
bits    * *
bits    * *
bits    * *
bits  *******
bits ** * * **
bits *  * *  **
bits *  * *  **
bits *  * *  **
bits ** * *
bits ***** *
bits  ******
bits   ******
bits     * ****
bits     * * ***
bits     * *  ***
bits     * *   **
bits     * *    *
bits ** * *    *
bits ** * *    *
bits ** * *   **
bits  ** * * ***
bits  ********
bits     * *
bits     * *
bits     * *
```

Blank lines and lines beginning with # are ignored. The first lines of the file form the font header. These are automatically set to their default values whenever *cfont* writes a binary font. The headers **hs**, **vs**, and **basel** specify the effective horizontal size, vertical size and baseline offset for the font as a whole. The rest of the file consists of up to 96 char definitions, each one beginning with the word char followed by the character number (0-95). After the character number has been specified, the various character definition fields are given followed by **vs** number of bits statements, each specifying exactly **hs** number of pixel columns.

SEE ALSO

font(4), window(7).

BUGS

It is currently impossible to specify the font flags or magic number.

NAME

chmod – change mode

SYNOPSIS

chmod mode files

DESCRIPTION

The permissions of the named *files* are 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**, the default if *who* is omitted.

Op can be + to add *permission* to the file's mode, – to take away *permission*, or = 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, or sticky); **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.

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** and **t** only works with **u**.

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

EXAMPLES

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

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

SEE ALSO

ls(1), chmod(2).

NAME

chown, chgrp – change owner or group

SYNOPSIS

chown owner file ...

chgrp group file ...

DESCRIPTION

Chown changes the owner of the *files* to *owner*. The owner may be either a decimal user ID or a login name found in the password file.

Chgrp changes the group ID of the *files* to *group*. The group may be either a decimal group ID or a group name found in the group file.

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(2), group(4), passwd(4).

NAME

clear – clear terminal screen

SYNOPSIS

clear

DESCRIPTION

Clear prints whatever string clears your terminal's screen. The program obtains this string from the *termcap*(5) database, using the **TERM** environment variable to determine the kind of terminal.

FILES

/etc/termcap terminal capability database

SEE ALSO

termcap(5)—terminal description database
sh(1)—**export** command

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] file1 file2

DESCRIPTION

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

Options:

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

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

SEE ALSO

comm(1), diff(1).

DIAGNOSTICS

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

NAME

col – filter reverse line-feeds

SYNOPSIS

col [**–bfpx**]

DESCRIPTION

Col reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse line feeds (ASCII code **ESC-7**), and by forward and reverse half-line-feeds (**ESC-9** and **ESC-8**). *Col* is particularly useful for filtering multicolumn output made with the **.rt** command of *nroff* and output resulting from use of the *tbl*(1) preprocessor.

If the **–b** option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

Although *col* accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. This treatment can be suppressed by the **–f** (fine) option; in this case, the output from *col* may contain forward half-line-feeds (**ESC-9**), but will still never contain either kind of reverse line motion.

Unless the **–x** option is given, *col* will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters **SO** (\017) and **SI** (\016) are assumed by *col* to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output **SI** and **SO** characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, new-line, **SI**, **SO**, **VT** (\013), and **ESC** followed by **7**, **8**, or **9**. The **VT** character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

Normally, *col* will ignore any unknown to it escape sequences found in its input; the **–p** option may be used to cause *col* to output these sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless the user is fully aware of the textual position of the escape sequences.

SEE ALSO

nroff(1), tbl(1).

NOTES

The input format accepted by *col* matches the output produced by *nroff* with either the **–T37** or **–Tlp** options. Use **–T37** (and the **–f** option of *col*) if the ultimate disposition of the output of *col* will be a device that can interpret half-line motions, and **–Tlp** otherwise.

BUGS

Cannot back up more than 128 lines.
Allows at most 800 characters, including backspaces, on a line.
Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [–o] [–s] [–psid] [–clist] files

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

–p*SID* The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

–c*list* A *list* (see *get*(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

–o For each **get** –**e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the –**o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

–s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

Comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

comm – select or reject lines common to two sorted files

SYNOPSIS

comm [– [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence (see *sort*(1)), and produces a three-column output: lines only in *file1*; lines only in *file2*; and lines in both files. The file name – means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm –12** prints only the lines common to the two files; **comm –23** prints only lines in the first file but not in the second; **comm –123** is a no-op.

SEE ALSO

cmp(1), diff(1), sort(1), uniq(1).

NAME

cp, ln, mv – copy, link or move files

SYNOPSIS

cp file1 [file2 ...] target
ln file1 [file2 ...] target
mv file1 [file2 ...] target

DESCRIPTION

File1 is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same (take care when using *sh*(1) metacharacters). If *target* is a directory, then one or more files are copied (linked, moved) to that directory.

If *mv* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)) and read the standard input for one line (if the standard input is a terminal); if the line begins with **y**, the move takes place; if not, *mv* exits.

Only *mv* will allow *file1* to be a directory, in which case the directory rename will occur only if the two directories have the same parent.

SEE ALSO

cpio(1), rm(1), chmod(2).

DIAGNOSTICS

Bad copy generally means I/O or other system error.

BUGS

If *file1* and *target* 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.

Ln will not link across file systems.

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio **–o** [**acvBTsize**Ooffset] < name-list >collection

cpio **–i** [**cdmrstuvRBTsize**Ooffset**f6**] [patterns] <collection

cpio **–p** [**adlmruv**] directory

DESCRIPTION

Cpio **–o** (copy out) reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information.

Cpio **–i** (copy in) extracts files from the standard input which is assumed to be the product of a previous **cpio** **–o**. Only files with names that match *patterns* are selected. *Patterns* are given in the name-generating notation of *sh*(1). In *patterns*, meta-characters **?**, *, and [. . .] match the slash / character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is * (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below.

When copying onto floppy diskettes, *cpio* records the number in sequence in the volume header of each floppy in the set. If in a subsequent *cpio* **–i** operation a floppy is loaded out of sequence, *cpio* pauses and prompts for the correct floppy.

Cpio **–p** (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination *directory* tree based upon the options described below.

The meanings of the available options are:

a Reset access times of input files after they have been copied.

B Input/output is to be blocked 5,120 bytes to the record.

d *Directories* are to be created as needed.

c Write *header* information in ASCII character form for portability.

K Reads only the first file of a cpio set. This option is used only with the **–i** option.

r Interactively *rename* files. If the user types a null line, the file is skipped.

R Allow files and directories with absolute path names to be redirected on input to the current working directory (see *pwd* (1)) by removing the leading / from the path name. This option is used only with the **–i** option.

O Set the logical file position of where the transfer is to begin. The **O** is followed by an *offset* in blocks. For example,

cpio -oc O128 > /dev/rfp021

will begin the output at block number 128.

t Print a *table of contents* of the input. No files are created.

T Provides a specific buffer size for the *cpio* operation. The size of the buffer, in kilobytes (**1KB = 1024B**), follows the **T**. For example,

cpio -ict T64 < /dev/rfp021

uses a **64KB** buffer while it reads the filenames from the *cpio* set. Note that if no buffer size is specified, 64KB buffers are used.

u Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).

v *Verbose*: causes a list of file names to be printed. When used with the **t** option, the table of contents looks like the output of an **ls -l** command (see *ls*(1)).

x Halt the *cpio* operation as soon as one *filename* in the pattern list is restored (otherwise the entire *cpio* set is read). This option is used only with the **-i** option.

l Whenever possible, link files rather than copying them. Usable only with the **-p** option.

m Retain previous file modification time. This option is ineffective on directories that are being copied.

f Copy in all files except those in *patterns*.

s Swap bytes. Use only with the **-i** option.

S Swap halfwords. Use only with the **-i** option.

b Swap both bytes and halfwords. Use only with the **-i** option.

6 Process an old (i.e., UNIX *Sixth* Edition format) file. Only useful with **-i** (copy in).

EXAMPLES

The first example below copies the contents of a directory into an archive; the second duplicates a directory hierarchy:

```
ls | cpio -o >/dev/rfp021

cd olddir
find . -depth -print | cpio -pdl newdir
```

The trivial case "find . -depth -print | cpio -oB >/dev/rfp021" can be handled more efficiently by:

```
find . -cpio /dev/rfp021
```

SEE ALSO

ar(1), find(1), cpio(4).

BUGS

Path names are restricted to 128 characters. If there are too many unique linked files, the program runs out of memory to keep

track of them and, thereafter, linking information is lost. Only the super-user can copy special files.

NAME

cpp – the C language preprocessor

SYNOPSIS

/lib/cpp [option ...] [ifile [ofile]]
/lib/mcpp -

DESCRIPTION

Cpp is the C language preprocessor which is invoked as the first pass of any C compilation using the *cc*(1) command. Thus the output of *cpp* is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(1) command since the functionality of *cpp* may someday be moved elsewhere. See *m4*(1) for a general macro processor.

Cpp optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

–P Preprocess the input without producing the line control information used by the next pass of the C compiler.

–C By default, *cpp* strips C-style comments. If the **–C** option is specified, all comments (except those found on *cpp* directive lines) are passed along.

–U*name*
Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The current list of these possibly reserved symbols includes:

operating system:	ibm, gcos, os, tss, unix
hardware:	interdata, pdp11, u370, u3b, vax, mc68K
UNIX variant:	RES, RT

–D*name*
–D*name*=*def*
Define *name* as if by a **#define** directive. If no =*def* is given, *name* is defined as 1.

–I*dir* Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in " " will be searched for first in the directory of the *ifile* argument, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *ifile* argument is not searched.

Two special names are understood by *cpp*. The name **__LINE__** is defined as the current line number (as a decimal integer) as known by *cpp*, and **__FILE__** is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by #. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string*.

#define *name*(*arg*, ..., *arg*) *token-string*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma separated tokens, and a) by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma separated list.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on.

#include *"filename"*
#include <*filename*>
Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the **-I** option above for more detail.

#line *integer-constant "filename"*
Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. If *"filename"* is not given, the current file name is unchanged.

#endif
Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**.

#ifdef *name*
The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*
The lines following will not appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#if *constant-expression*
Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, the unary **-**, **!**, and **~** operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by

the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined** (*name*) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

#else Reverses the notion of the test directive which matches this directive. So if lines previous to this directive are ignored, the following lines will appear in the output. And vice versa.

The test directives and the possible **#else** directives can be nested.

FILES

/usr/include standard directory for **#include** files

SEE ALSO

cc(1), m4(1).

DIAGNOSTICS

The error messages produced by *cpp* are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NOTES

When newline characters were found in argument lists for macros to be expanded, previous versions of *cpp* put out the newlines as they were found and expanded. The current version of *cpp* replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

NAME

crypt – encode/decode

SYNOPSIS

crypt [password]

DESCRIPTION

This command is available only in the domestic (U.S.) version of the UNIX PC software. *Crypt* reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

will print the clear.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; "sneak paths" by which keys or clear text can become visible must be minimized.

Crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e. to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lower-case letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. To minimize this possibility, *crypt* takes care to destroy any record of the key immediately upon entry. The choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(1).

BUGS

If output is piped to *nroff* and the encryption key is *not* given on the command line, *crypt* can leave terminal modes in a strange state (see *stty*(1)).

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

NAME

csplit – context split

SYNOPSIS

csplit [**–s**] [**–k**] [**–f** prefix] file arg1 [**. . .** argn]

DESCRIPTION

Csplit reads *file* and separates it into n+1 sections, defined by the arguments *arg1. . . argn*. By default the sections are placed in xx00 **. . .** xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.

⋮

n+1: From the line referenced by *argn* to the end of *file*.

The options to *csplit* are:

–s *Csplit* normally prints the character counts for each file created. If the **–s** option is present, *csplit* suppresses the printing of all character counts.

–k *Csplit* normally removes created files if an error occurs. If the **–k** option is present, *csplit* leaves previously created files intact.

–f *prefix* If the **–f** option is used, the created files are named *prefix***00 . . .** *prefixn*. The default is **xx00 . . . xx***n*.

The arguments (*arg1* **. . .** *argn*) to *csplit* can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or – some number of lines (e.g., **/Page/–5**).

%rexp% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the Shell in the appropriate quotes.

Regular expressions may not contain embedded new-lines. *Csplit* does not affect the original file; it is the users responsibility to remove it.

EXAMPLES

csplit –f cobol file '/procedure division/' /par5./ /par16./

This example creates four files, **cobol00 . . . cobol03**. After editing the "split" files, they can be recombined as follows:

cat cobol0[0–3] > file

Note that this example overwrites the original file.

csplit –k file 100 {99}

This example would split the file at every 100 lines, up to 10,000 lines. The **–k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

csplit –k prog.c '%main(%' '/^}/+1' {20}

Assuming that **prog.c** follows the normal **C** coding convention of ending routines with a **}** at the beginning of the line, this example will create a file containing each separate **C** routine (up to 21) in **prog.c**.

SEE ALSO

ed(1), sh(1), regexp(5).

DIAGNOSTICS

Self explanatory except for:

arg – out of range

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

cu – call another UNIX system

SYNOPSIS

cu [**–s**speed] [**–l**line] [**–h**] [**–t**] [**–d**] [**–m**] [**–o**| **–e**] telno | **dir**

DESCRIPTION

Cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files. *Speed* gives the transmission speed (110, 150, 300, 600, 1200, 4800, 9600); 300 is the default value. Most of our modems are either 300 or 1200 baud. For dial out lines, *cu* will choose a modem speed (300 or 1200) as the slowest available which will handle the specified transmission speed. Directly connected lines may be set to speeds higher than 1200 baud.

The –l value may be used to specify a device name for the communications line device to be used. This can be used to override searching for the first available line having the right speed. The -s option allows the user to override the line speed specified in the file /usr/lib/uucp/L-devices. However, if the -s option is not used, the line speed will be taken from the L-devices file. The **–h** option emulates local echo, supporting calls to other computer systems which expect terminals to be in half-duplex mode. The **–t** option is used when dialing an ASCII terminal which has been set to auto-answer. Appropriate mapping of carriage-returns to carriage-return-line-feed pairs is set. The **–d** option cause diagnostic traces to be printed. The **–m** option specifies a direct line which has modem control. The **–e** (**–o**) option designates that even (odd) parity is to be generated for data sent to the remote. The **–d** option causes diagnostic traces to be printed. *Telno* is the telephone number, with '=' (equal signs) for secondary dial tone. ':' (colons) for pausing 10 seconds, and for pausing 2 seconds at appropriate places. The string **dir** for *telno* may be used for directly connected lines, and implies a null ACU. Using **dir** insures that a line has been specified by the **-l** option. When using the internal modem line, ph0 and ph1 make sure the phone status of the line to be used shows DATA, otherwise the call will fail. The phone line supports 300 and 1200 for the -s option.

Cu will try each line listed in the file */usr/lib/uucp/L-devices* until it finds an available line with appropriate attributes or runs out of entries. After making the connection, *cu* runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following:

~.	terminate the conversation.
~!	escape to an interactive shell on the local system.
~!*cmd* . . .	run *cmd* on the local system (via **sh** −**c**).
~$*cmd* . . .	run *cmd* locally and send its output to the remote system.
~%**take** *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~%**put** *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places.
~~. . .	send the line ~. . . to the remote system.
~%**nostop**	turn off the DC3/DC1 input control protocol for the remainder of the session. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters,

The *receive* process normally copies data from the remote system to its standard output. A line from the remote that begins with ~> initiates an output diversion to a file. The complete sequence is:

~>[>]: *file*
zero or more lines to be written to *file*
~>

Data from the remote is diverted (or appended, if >> is used) to file. The trailing ~> terminates the diversion.

The use of ~%**put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of ~%**take** requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, **stty tabs** mode should be set on the remote system if tabs are to be copied without expansion.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..(tty-device)
/dev/null

SEE ALSO

cat(1), echo(1), stty(1), uucp(1C).

DIAGNOSTICS

Exit code is zero for normal exit, non-zero (various values) otherwise.

BUGS

Cu buffers input internally.
There is an artificial slowing of transmission by *cu* during the ~%**put** operation so that loss of data is unlikely.

NAME

cut – cut out selected fields of each line of a file

SYNOPSIS

cut **–c**list [file1 file2 ...]
cut **–f**list [**–d** char] [**–s**] [file1 file2 ...]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (**–c** option), or the length can vary from line to line and be marked with a field delimiter character like *tab* (**–f** option). *Cut* can be used as a filter; if no files are given, the standard input is used.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional – to indicate ranges as in the **–o** option of *nroff/troff* for page ranges; e.g., **1,4,7**; **1–3,8**; **–5,10** (short for **1–5,10**); or **3–** (short for third through last field).

–c *list* The *list* following **–c** (no space) specifies character positions (e.g., **–c1–72** would pass the first 72 characters of each line).

–f *list* The *list* following **–f** is a list of fields assumed to be separated in the file by a delimiter character (see **–d**); e.g. , **–f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless **–s** is specified.

–d *char* The character following **–d** is the field delimiter (**–f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

–s Suppresses lines with no delimiter characters in case of **–f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the **–c** or **–f** option must be specified.

HINTS

Use *grep*(1) to make horizontal "cuts" (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut –d: –f1,5 /etc/passwd	mapping of user IDs to names
name=`who am i | cut –f1 –d" "`	to set **name** to current login name.

DIAGNOSTICS

line too long A line can have no more than 511 characters or fields.

bad list for c / f option Missing **-c** or **-f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

no fields The *list* is empty.

SEE ALSO

grep(1), paste(1).

NAME

cw, checkcw – prepare constant-width text for troff

SYNOPSIS

cw [**-lxx**] [**-rxx**] [**-fn**] [**-t**] [**+t**] [**-d**] [files]

checkcw [**-lxx**] [**-rxx**] files

DESCRIPTION

Cw is a preprocessor for *troff* (not included on the UNIX PC) input files that contain text to be typeset in the constant-width (CW) font.

Text typeset with the CW font resembles the output of terminals and of line printers. This font is used to typeset examples of programs and of computer output in user manuals, programming texts, etc. (An earlier version of this font was used in typesetting *The C Programming Language* by B. W. Kernighan and D. M. Ritchie.) It has been designed to be quite distinctive (but not overly obtrusive) when used together with the Times Roman font.

Because the CW font contains a "non-standard" set of characters and because text typeset with it requires different character and inter-word spacing than is used for "standard" fonts, documents that use the CW font must be preprocessed by *cw*.

The CW font contains the 94 printing ASCII characters:

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!$%&()"*+@.,/:;=?[]|-_^~"<>{}#\

plus eight non-ASCII characters represented by four-character *troff* names (in some cases attaching these names to "non-standard" graphics):

Character	Symbol	Troff Name
"Cents" sign		\(ct
EBCDIC "not" sign	¬	\(no
Left arrow	←	\(<-
Right arrow	→	\(->
Down arrow	↓	\(da
Vertical single quote	'	\(fm
Control-shift indicator	†	\(dg
Visible space indicator		\(sq
Hyphen	-	\(hy

The hyphen is a synonym for the unadorned minus sign (**-**). Certain versions of *cw* recognize two additional names: **\(ua** for an up arrow (↑) and **\(lh** for a diagonal left-up (home) arrow.

Cw recognizes five request lines, as well as user-defined delimiters. The request lines look like *troff* macro requests, and are copied in their entirety by *cw* onto its output; thus, they can be defined *by the user* as *troff* macros; in fact, the .CW and .CN macros *should* be so defined (see *HINTS* below). The five requests are:

.CW Start of text to be set in the CW font; **.CW** causes a break; it can take precisely the same options, in precisely the same format, as are available on the *cw* command line.

.CN End of text to be set in the CW font; **.CN** causes a break; it can take the same options as are available on the *cw* command line.

.CD Change delimiters and/or settings of other options; takes the same options as are available on the *cw* command line.

.CP *arg1 arg2 arg3 ... argn*
All the arguments (which are delimited like *troff* macro arguments) are concatenated, with the odd-numbered arguments set in the CW font and the even-numbered ones in the prevailing font.

.PC *arg1 arg2 arg3 ... argn*
Same as **.CP**, except that the even-numbered arguments are set in the CW font and the odd-numbered ones in the prevailing font.

The .CW and .CN requests are meant to bracket text (e.g., a program fragment) that is to be typeset in the CW font "as is." Normally, *cw* operates in the *transparent* mode. In that mode, except for the .CD request and the nine special four-character names listed in the table above, every character between .CW and .CN request lines stands for itself. In particular, *cw* arranges for periods (.) and apostrophes (') at the beginning of lines, and backslashes (\) everywhere to be "hidden" from *troff*. The transparent mode can be turned off (see below), in which case normal *troff* rules apply; in particular, lines that begin with . and ' are passed through untouched (except if they contain delimiters–see below). In either case, *cw* hides the effect of the font changes generated by the .CW and .CN requests; *cw* also defeats all ligatures (fi, ff, etc.) in the CW font.

The only purpose of the .CD request is to allow the changing of various options other than just at the beginning of a document.

The user can also define *delimiters*. The left and right delimiters perform the same function as the .CW /.CN requests; they are meant, however, to enclose CW "words" or "phrases" in running text (see example under *BUGS* below). *Cw* treats text between delimiters in the same manner as text enclosed by .CW /.CN pairs, except that, for aesthetic reasons, spaces and backspaces inside .CW /.CN pairs have the same width as other CW characters, while spaces and backspaces between delimiters are half as wide, so they have the same width as spaces in the prevailing text (but are *not* adjustable). Font changes due to delimiters are *not* hidden.

Delimiters have no special meaning inside .CW /.CN pairs.

The options are:

−l*xx* The one- or two-character string *xx* becomes the left delimiter; if *xx* is omitted, the left delimiter becomes undefined, which it is initially.

−r*xx* Same for the right delimiter. The left and right delimiters may (but need not) be different.

−f*n* The CW font is mounted in font position *n*; acceptable values for *n* are 1, 2, and 3 (default is 3, replacing the bold font). This option is only useful at the beginning of a document.

−t Turn transparent mode *off*.

+t Turn transparent mode *on* (this is the initial default).

−d Print current option settings on file descriptor 2 in the form of *troff* comment lines. This option is meant for debugging.

Cw reads the standard input when no *files* are specified (or when − is specified as the last argument), so it can be used as a filter. Typical usage is:

cw *files* | troff ...

Checkcw checks that left and right delimiters, as well as the .CW /.CN pairs, are properly balanced. It prints out all offending lines.

HINTS

Typical definitions of the .CW and .CN macros meant to be used with the *mm*(5) macro package:

```
.de CW
.DS I
.ps 9
.vs 10.5p
.ta 16m/3u 32m/3u 48m/3u 64m/3u 80m/3u 96m/3u ...
..
.de CN
.ta .5i 1i 1.5i 2i 2.5i 3i ...
.vs
.ps
.DE
..
```

At the very least, the .CW macro should invoke the *troff* no-fill (**.nf**) mode.

When set in running text, the CW font is meant to be set in the same point size as the rest of the text. In displayed matter, on the other hand, it can often be profitably set one point *smaller* than the prevailing point size (the displayed definitions of .CW and .CN above are one point smaller than the running text on this page). The CW font is sized so that, when it is set in 9-point, there are 12 characters per inch.

Documents that contain CW text may also contain tables and/or equations. If this is the case, the order of preprocessing should be: *cw*, *tbl*, and *eqn*. Usually, the tables contained in such documents will not contain any CW text, although it is entirely possible to have *elements* of the table set in the CW font; of course, care must be taken that *tbl*(1) format information not be modified by *cw*. Attempts to set equations in the CW font are not likely to be either pleasing or successful.

In the CW font, overstriking is most easily accomplished with backspaces: letting ← represent a backspace, d←←† yields ₫. (Because backspaces are half as wide between delimiters as inside .CW /.CN pairs–see above–two backspaces are required for each overstrike between delimiters.)

FILES

/usr/lib/font/ftCW CW font-width table

SEE ALSO

eqn(1), mmt(1), tbl(1), mm(5).

WARNINGS

If text preprocessed by *cw* is to make any sense, it must be set on a typesetter equipped with the CW font or on a STARE facility; on the latter, the CW font appears as bold, but with the proper CW spacing.

BUGS

Only a masochist would use periods (.), backslashes (\), or double quotes (") as delimiters, or as arguments to .CP and .PC.

Certain CW characters don't concatenate gracefully with certain Times Roman characters, e.g., a CW ampersand (&) followed by a Times Roman comma (,); in such cases, judicious use of *troff* half- and quarter-spaces (\| and \^) is most salutary, e.g., one should use **_&_\^,** (rather than just plain **_&_,**) to obtain **&,** (assuming that _ is used for both delimiters).

Using *cw* with *nroff* is silly.

The output of *cw* is hard to read.

NAME

cxref – generate C program cross reference

SYNOPSIS

cxref [options] files

DESCRIPTION

Cxref analyzes a collection of C files and attempts to build a cross reference table. *Cxref* utilizes a special version of *cpp* to include **#define**'d information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or with the **–c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **–D**, **–I** and **–U** options (which are identical to their interpretation by *cc*(1)), the following *options* are interpreted by *cxref*:

–c Print a combined cross-reference of all input files.

–w<num>
Width option which formats output no wider than <num> (decimal) columns. This option will default to 80 if <num> is not specified or is less than 51.

–o file Direct output to named *file*.

–s Operate silently; does not print input file names.

–t Format listing for 80-column width.

FILES

/usr/lib/xcpp special version of C-preprocessor.

SEE ALSO

cc(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you can't compile these files, anyway.

NAME

date – print and set the date

SYNOPSIS

date [mmddhhmm[yy]] [+format]
date -

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set. 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; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

If the argument begins with +, the output of *date* is under the control of the user. The format for the output is similar to that of the first argument to *printf*(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a new-line character.

Field Descriptors:

n	insert a new-line character
t	insert a tab character
m	month of year – 01 to 12
d	day of month – 01 to 31
y	last 2 digits of year – 00 to 99
D	date as mm/dd/yy
H	hour – 00 to 23
M	minute – 00 to 59
S	second – 00 to 59
T	time as HH:MM:SS
j	day of year – 001 to 366
w	day of week – Sunday = 0
a	abbreviated weekday – Sun to Sat
h	abbreviated month – Jan to Dec
r	time in AM/PM notation

date - sets the system time from the real time clock.

EXAMPLE

date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'

would have generated as output:

DATE: 08/01/76
TIME: 14:45:05

DIAGNOSTICS

No permission	if you aren't the super-user and you try to change the date;
bad conversion	if the date set is syntactically incorrect;
bad format character	if the field descriptor is not recognizable.

FILES

/dev/kmem

WARNING

It is a bad practice to change the date while the system is running multi-user.

NAME

dc – desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
: The value of the number is pushed on the stack. A number is an unbroken string of the digits 0–9. It may be preceded by an underscore (_) to input a negative number. Numbers may contain decimal points.

\+ – / * % ^
: The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x*
: The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x*
: The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d
: The top value on the stack is duplicated.

p
: The top value on the stack is printed. The top value remains unchanged. **P** interprets the top of the stack as an ASCII string, removes it, and prints it.

f
: All values on the stack are printed.

q
: exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x
: treats the top element of the stack as a character string and executes it as a string of *dc* commands.

X
: replaces the number on the top of the stack with its scale factor.

[...]
: puts the bracketed ASCII string onto the top of the stack.

<x **>x** **=x**
The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation.

v replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! interprets the rest of the line as a UNIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. **I** pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O pushes the output base on the top of the stack.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : are used by *bc* for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1), which is a preprocessor for *dc* providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

x is unimplemented
where *x* is an octal number.

stack empty
for not enough elements on the stack to do what was asked.

Out of space
when the free list is exhausted (too many digits).

Out of headers
for too many numbers being kept around.

Out of pushdown
for too many items on the stack.

Nesting Depth
for too many levels of nested execution.

NAME

dd – convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

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

option	*values*
if=*file*	input file name; standard input is default
of=*file*	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 in-core copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
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.

EXAMPLE

This command will read an EBCDIC floppy blocked ten 80-byte EBCDIC card images per record into the ASCII file **x** :

```
dd if=/dev/rfp021 of=x ibs=800 cbs=80 conv=ascii,lcase
```

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

SEE ALSO

cp(1).

DIAGNOSTICS

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

BUGS

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

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

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [**–r**SID] [**–s**] [**–n**] [**–g**list] [**–m**[mrlist]] [**–y**[comment]] [**–p**] files

DESCRIPTION

Delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

Delta makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

Delta may issue prompts on the standard output depending upon certain keyletters specified and flags (see *admin*(1)) that may be present in the SCCS file (see **–m** and **–y** keyletters below).

Keyletter arguments apply independently to each named file.

–r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (**get –e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **–r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(1)). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

–n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

–g*list* Specifies a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

–m[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(1)) then a Modification Request (**MR**) number *must* be supplied as the reason for creating the new delta.

If **−m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see **−y** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the **MR** list.

Note that if the **v** flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the **MR** numbers. If a non-zero exit status is returned from **MR** number validation program, *delta* terminates (it is assumed that the **MR** numbers were not all valid).

−y[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

If **−y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text.

−p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

All files of the form *?*-file are explained in the *Source Code Control System User's Guide*. The naming convention for these files is also described there.

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

WARNINGS

Lines beginning with an **SOH** ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS (see *sccsfile*(5)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (–) is specified on the *delta* command line, the **–m** (if necessary) and **–y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

SEE ALSO

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

deroff – remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [**–mx**] [**–w**] [files]

DESCRIPTION

Deroff reads each of the *files* in sequence and removes all *troff* requests, macro calls, backslash constructs, *eqn*(1) constructs (between **.EQ** and **.EN** lines, and between delimiters), and *tbl*(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output. *Deroff* follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The **–m** option may be followed by an **m**, **s**, or **l**. The **–mm** option causes the macros be interpreted so that only running text is output (i.e., no text from macro lines.) The **–ml** option forces the **–mm** option and also causes deletion of lists associated with the **mm** macros.

If the **–w** option is given, the output is a word list, one "word" per line, with all other characters deleted. Otherwise, the output follows the original, with the deletions mentioned above. In text, a "word" is any string that *contains* at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a "word" is a string that *begins* with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from "words."

SEE ALSO

eqn(1), nroff(1), tbl(1).

BUGS

Deroff is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most such errors result in too much rather than too little output.

The **–ml** option does not handle nested lists correctly.

NAME

diff – differential file and directory comparator

SYNOPSIS

diff [**-l**] [**-r**] [**-s**] [**-cefh**] [**-b**] dir1 dir2
diff [**-cefh**] [**-b**] file1 file2
diff [**-D***string*] [**-b**] file1 file2

DESCRIPTION

If both arguments are directories, *diff* sorts the contents of the directories by name, and then runs the regular file *diff* algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed. Options when comparing directories are:

-l long output format; each text file *diff* is piped through *pr*(1) to paginate it, other differences are remembered and summarized after all text file differences are reported.

-r causes application of *diff* recursively to common subdirectories encountered.

-s causes *diff* to report files which are the same, which are otherwise not mentioned.

-Sname
starts a directory *diff* in the middle beginning with file *name*.

When run on regular files, and when comparing text files which differ during directory comparison, *diff* tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, *diff* finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, then either may be given as '-', in which case the standard input is used. If *file1* is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* is used (and vice versa).

There are several options for output format; the default output format contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where $n1 = n2$ or $n3 = n4$ are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

Except for **-b**, which may be given with any of the others, the following options are mutually exclusive:

-e producing a script of *a, c* and *d* commands for the editor *ed,* which will recreate *file2* from *file1*. In connection with **-e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Extra commands are added to the output when comparing directories with **-e,** so that the result is a *sh*(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2.*

-f produces a script similar to that of **-e,** not useful with *ed,* and in the opposite order.

-c produces a diff with lines of context. The default is to present 3 lines of context and may be changed, e.g to 10, by **-c10**. With **-c** the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with '-'; those added to *file2* are marked '+'. Lines which are changed from one file to the other are marked in both files with '!'.

-h does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

-Dstring
causes *diff* to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1,* while defining *string* will yield *file2.*

-b causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

FILES

/tmp/d?????
/usr/lib/diffh for **-h**
/bin/pr

SEE ALSO

cmp(1), cc(1), comm(1), ed(1), diff3(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

BUGS

Editing scripts produced under the **-e** or **-f** option are naive about creating lines consisting of a single '.'

When comparing directories with the **-b** option specified, *diff* first compares the files ala *cmp*, and then decides to run the *diff* algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical because the only differences are insignificant blank string differences.

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**–ex3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f **:** *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f **:** *n1* **,** *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **–e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, i.e., the changes that normally would be flagged ==== and ====3. Option **–x** (**–3**) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

(cat script; echo '1,$p') | ed – file1

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1).

BUGS

Text lines that consist of a single . will defeat **–e**.
Files longer than 64K bytes won't work.

NAME

diffmk – mark differences between files

SYNOPSIS

diffmk name1 name2 name3

DESCRIPTION

Diffmk compares two versions of a file and creates a third file that includes "change mark" commands for *nroff* or *troff*. *Name1* and *name2* are the old and new versions of the file. *Diffmk* generates *name3*, which contains the lines of *name2* plus inserted formatter "change mark" (**.mc**) requests. When *name3* is formatted, changed or inserted text is shown by | at the right margin of each line. The position of deleted text is shown by a single *.

If anyone is so inclined, *diffmk* can be used to produce listings of C (or other) programs with changes marked. A typical command line for such use is:

```
diffmk old.c new.c tmp; nroff macs tmp | pr
```

where the file **macs** contains:

```
.pl 1
.ll 77
.nf
.eo
.nc ‘
```

The **.ll** request might specify a different line length, depending on the nature of the program being printed. The **.eo** and **.nc** requests are probably needed only for C programs.

If the characters | and * are inappropriate, a copy of *diffmk* can be edited to change them (*diffmk* is a shell procedure).

SEE ALSO

diff(1), nroff(1).

BUGS

Aesthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output, i.e., replacing **.sp** by **.sp 2** will produce a "change mark" on the preceding or following line of output.

NAME

dircmp – directory comparison

SYNOPSIS

dircmp [**–d**] [**–s**] dir1 dir2

DESCRIPTION

Dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

–d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

–s Suppress messages about identical files.

SEE ALSO

cmp(1), diff(1).

NAME

du – summarize disk usage

SYNOPSIS

du [**–ars**] [names]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each directory and file specified by the *names* argument. The block count includes the indirect blocks of the file. If *names* is missing, **.** is used.

The optional argument **–s** causes only the grand total (for each of the specified *names*) to be given. The optional argument **–a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

Du is normally silent about directories that cannot be read, files that cannot be opened, etc. The **–r** option will cause *du* to generate messages in such instances.

A file with two or more links is only counted once.

BUGS

If the **–a** option is not used, non-directories given as arguments are not listed.
If there are too many distinct linked files, *du* will count the excess files more than once.
Files with holes in them will get an incorrect block count.

NAME

dump – dump selected parts of an object file

SYNOPSIS

dump [**–a**] [**–f**] [**–o**] [**–h**] [**–s**] [**–r**] [**–l**] [**–t**] [**–z** name] files

DESCRIPTION

The *dump* command dumps selected parts of each of its object *file* arguments.

This command will accept both object files and archives of object files. It processes each file argument according to one or more of the following options:

–a Dump the archive header of each member of each archive file argument.

–f Dump each file header.

–o Dump each optional header.

–h Dump section headers.

–s Dump section contents.

–r Dump relocation information.

–l Dump line number information.

–t Dump symbol table entries.

–z name Dump line number entries for the named function.

The following *modifiers* are used in conjunction with the options listed above to modify their capabilities.

–d number Dump the section number or range of sections starting at *number* and ending either at the last section number or *number* specified by **+d**.

+d number Dump sections in the range either beginning with first section or beginning with section specified by **–d**.

–n name Dump information pertaining only to the named entity. This *modifier* applies to **–h**, **–s**, **–r**, **–l**, and **–t**.

–t index Dump only the indexed symbol table entry. The **–t** used in conjunction with **+t**, specifies a range of symbol table entries.

+t index Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **–t** option.

–v Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of **0X02**). This *modifier* can be used with all the above options except **–s** and **–o** options of *dump*.

−z name,number

Dump line number entry or range of line numbers starting at *number* for the named function.

+z number

Dump line numbers starting at either function *name* or *number* specified by **−z**, up to *number* specified by **+z**.

Blanks separating an *option* and its *modifier* are optional. The comma separating the name from the number modifying the **−z** option may be replaced by a blank.

The *dump* command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal or decimal representation as appropriate.

SEE ALSO

a.out(4), ar(4).

NAME

echo – echo arguments

SYNOPSIS

echo [arg] ...

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a new-line on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b	backspace
\c	print line without new-line
\f	form-feed
\n	new-line
\r	carriage return
\t	tab
****	backslash
\n	the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a zero.

Echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

NAME

ed, red – text editor

SYNOPSIS

ed [–] [**–x**] [file]

red [–] [**–x**] [file]

DESCRIPTION

Ed is the standard text editor. If the *file* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited. The optional – suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the ! prompt after a !*shell command*. If **–x** is present, an *x* command is simulated first to handle an encrypted file; this capability is present only in the domestic (U.S) version of the UNIX PC software. *Ed* operates on a copy of the file it is editing; changes made to 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.

Red is a restricted version of *ed*. It will only allow editing of files in the current directory. It prohibits executing shell commands via !*shell command*. Attempts to bypass these restrictions result in an error message (*restricted shell*).

Both *ed* and *red* support the *fspec*(4) formatting capability. After including a format specification as the first line of *file* and invoking *ed* with your terminal in **stty –tabs** or **stty tab3** mode (see *stty*(1), the specified tab stops will automatically be used when scanning *file*. For example, if the first line of a file contained:

<:t5,10,15 s72:>

tab stops would be set at columns 5, 10 and 15, and a maximum line length of 72 would be imposed. NOTE: while inputting text, tab characters when typed are expanded to every eighth column as is the default.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very 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 *ed* 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.

Ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be *matched* by

the RE. The REs allowed by *ed* are constructed as follows:

The following *one-character RE*s match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets ([]; see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([]) (see 1.4 below).

c. $ (currency symbol), which is special at the *end* of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the *g* command, below.)

1.3 A period (.) is a one-character RE that matches any character except new-line.

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character *except* new-line and the remaining characters in the string. The ^ has this special meaning *only* if it occurs first in the string. The minus (–) may be used to indicate a range of consecutive ASCII characters; for example, **[0–9]** is equivalent to **[0123456789]**. The – loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); e.g., **[]a–f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; \{*m*\} matches *exactly m* occurrences; \{*m*,\} matches *at least m* occurrences; \{*m*,*n*\} matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both):

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a *final* segment of a line.

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (e.g., //) is equivalent to the last RE encountered. See also the last paragraph before *FILES* below.

To understand addressing in *ed* 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; the exact effect on the current line is discussed under the description of each 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 RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before

FILES below.

6. A RE enclosed in question marks (**?**) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before *FILES* below.
7. An address followed by a plus sign (+) or a minus sign (−) followed by a decimal number specifies that address plus (respectively 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 to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address − refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to −.) Moreover, trailing + and − characters have a cumulative effect, so −− refers to the current line less 2.
10. For convenience, a comma (**,**) stands for the address pair **1,$**, while a semicolon (**;**) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (**,**). They may also be separated by a semicolon (**;**). In the latter case, the current line (**.**) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **l**, **n** or **p**, in which case the current line is either listed, numbered or printed, respectively, as discussed below under the *l*, *n* and *p* commands.

(.)a

<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the "appended" text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

(.,.)d

The *d*elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

The *e*dit 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. If no file name is given, the currently-remembered file name, if any, is used (see the *f* command). The number of characters read is typed; *file* is remembered for possible use as a default file name in subsequent *e*, *r*, and *w* commands. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. Such a shell command is *not* remembered as the current file name. See also *DIAGNOSTICS* below.

E *file*

The *E*dit command is like *e*, except that the editor does not check to see if any changes have been made to the buffer since the last *w* command.

f *file*

If *file* is given, the *f*ile-name command changes the currently-remembered file name to *file*; otherwise, it prints the currently-remembered file name.

(1,$)g/*RE***/***command list*

In the *g*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with . initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; *a*, *i*, and *c* commands and associated input are permitted; the . terminating input mode may be omitted if it would be the last line of the *command list*. An empty

command list is equivalent to the *p* command. The *g*, *G*, *v*, and *V* commands are *not* permitted in the *command list*. See also *BUGS* and the last paragraph before *FILES* below.

(1,$)G/*RE*/

In the interactive *G*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, **.** is changed to that line, and any *one* command (other than one of the *a*, *c*, *i*, *g*, *G*, *v*, and *V* commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a new-line acts as a null command; an *&* causes the re-execution of the most recent command executed within the current invocation of *G*. Note that the commands input as part of the execution of the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

The *h*elp command gives a short error message that explains the reason for the most recent **?** diagnostic.

H

The *H*elp command causes *ed* to enter a mode in which error messages are printed for all subsequent **?** diagnostics. It will also explain the previous **?** if there was one. The *H* command alternately turns this mode on and off; it is initially off.

(.)i
<text>
.

The *i*nsert command inserts the given text before the addressed line; **.** is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the *a* command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be a lower-case letter. The address *'x* then addresses this line; **.** is unchanged.

(.,.)l

The *l*ist command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., *tab*, *backspace*) are represented by (hopefully) mnemonic overstrikes, all other non-printing characters are printed in

octal, and long lines are folded. An *l* command may be appended to any other command other than *e*, *f*, *r*, or *w*.

(.,.)m*a*

The *m*ove command repositions the addressed line(s) after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file; it is an error if address *a* falls within the range of moved lines; . is left at the last line moved.

(.,.)n

The *n*umber command prints the addressed lines, preceding each line by its line number and a tab character; . is left at the last line printed. The *n* command may be appended to any other command other than *e*, *f*, *r*, or *w*.

(.,.)p

The *p*rint command prints the addressed lines; . is left at the last line printed. The *p* command may be appended to any other command other than *e*, *f*, *r*, or *w*; for example, *dp* deletes the current line and prints the new current line.

P

The editor will prompt with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.

q

The *q*uit command causes *ed* to exit. No automatic write of a file is done (but see *DIAGNOSTICS* below).

Q

The editor exits without checking if changes have been made in the buffer since the last *w* command.

($)r *file*

The *r*ead command reads in the given file after the addressed line. If no file name is given, the currently-remembered file name, if any, is used (see *e* and *f* commands). The currently-remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. 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 set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. For example, "$r !ls" appends current directory to the end of the file being edited. Such a shell command is *not* remembered as the current file name.

(.,.)s/*RE*/*replacement*/ or
(.,.)s/*RE*/*replacement*/g

The *s*ubstitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) matched strings are

replaced by the *replacement* 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 RE and the *replacement*; . is left at the last line on which a substitution occurred. See also the last paragraph before *FILES* below.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. 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 of the specified RE enclosed between \(and \). When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

A line may be split by substituting a new-line character into it. The new-line in the *replacement* must be escaped by preceding it by \. Such substitution cannot be done as part of a *g* or *v* command list.

(.,.)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 at the last line of the copy.

u

The *u*ndo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent *a*, *c*, *d*, *g*, *i*, *j*, *m*, *r*, *s*, *t*, *v*, *G*, or *V* command.

(1,$)v/*RE*/*command list*

This command is the same as the global command *g* except that the *command list* is executed with . initially set to every line that does *not* match the RE.

(1,$)V/*RE*/

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do *not* match the RE.

(1,$)w *file*

The *w*rite command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your *umask* setting (see *sh*(1)) dictates otherwise. The

currently-remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. If no file name is given, the currently-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. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current file name.

X

This command is available only in the domestic (U.S.) version of the UNIX PC software. A key string is demanded from the standard input. Subsequent *e*, *r*, and *w* commands will encrypt and decrypt the text with this key by the algorithm of *crypt*(1). An explicitly empty key turns off encryption.

(.)=

The line number of the addressed line is typed; . is unchanged by this command.

!*shell command*

The remainder of the line after the ! is sent to the UNIX shell (*sh*(1)) to be interpreted as a command. Within the text of that command, the unescaped character % is replaced with the remembered file name; if a ! appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, !! will repeat the last shell command. If any expansion is performed, the expanded line is echoed; . is unchanged.

(.+1)<new-line>

An address alone on a line causes the addressed line to be printed. A new-line alone is equivalent to **.+1p**; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII DEL or BREAK) is sent, *ed* 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 buffer. The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

When reading a file, *ed* discards ASCII NUL characters and all characters after the last new-line. Files (e.g., **a.out**) that contain characters not in the ASCII set (bit 8 on) cannot be edited by *ed*.

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a new-line, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

s/s1/s2	s/s1/s2/p
g/s1	g/s1/p
?s1	?s1?

FILES

/tmp/e#	temporary; # is the process number.
ed.hup	work is saved here if the terminal is hung up.

DIAGNOSTICS

?	for command errors.
?*file*	for an inaccessible file. (use the *h*elp and *H*elp commands for detailed explanations).

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy *ed*'s buffer via the *e* or *q* commands: it prints ? and allows one to continue editing. A second *e* or *q* command at this point will take effect. The – command-line option inhibits this feature.

SEE ALSO

crypt(1), grep(1), sed(1), sh(1), stty(1), fspec(4), regexp(5).
A Tutorial Introduction to the UNIX Text Editor by B. W. Kernighan.
Advanced Editing on UNIX by B. W. Kernighan.

CAVEATS AND BUGS

A *!* command cannot be subject to a *g* or a *v* command.
The *!* command and the **!** escape from the *e*, *r*, and *w* commands cannot be used if the the editor is invoked from a restricted shell (see *sh*(1)).
The sequence **\n** in a RE does not match a new-line character.
The *l* command mishandles DEL.
Files encrypted directly with the *crypt*(1) command with the null key cannot be edited.
Characters are masked to 7 bits on input.

NAME

enable, disable – enable/disable LP printers

SYNOPSIS

enable printers
disable [**-c**] [**-r**[reason]] printers

DESCRIPTION

Enable activates the named *printers*, enabling them to print requests taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

Disable deactivates the named *printers*, disabling them from printing requests taken by *lp*(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use *lpstat*(1) to find the status of printers. Options useful with *disable* are:

-c Cancel any requests that are currently printing on any of the designated printers.

-r[*reason*] Associates a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next **-r** option. If the **-r** option is not present or the **-r** option is given without a reason, then a default reason will be used. *Reason* is reported by *lpstat*(1).

FILES

/usr/spool/lp/*

SEE ALSO

lp(1), lpstat(1).

NAME

env – set environment for command execution

SYNOPSIS

env [–] [name=value] ... [command args]

DESCRIPTION

Env obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

SEE ALSO

sh(1), exec(2), profile(4), environ(5).

NAME

eqn, neqn, checkeq – format mathematical text for nroff or troff

SYNOPSIS

eqn [**–dxy**] [**–pn**] [**–sn**] [**–fn**] [files]

neqn [**–dxy**] [**–pn**] [**–sn**] [**–fn**] [files]

checkeq [files]

DESCRIPTION

Eqn is a *troff* preprocessor for typesetting mathematical text on a phototypesetter, while *neqn* is used for the same purpose with *nroff* on typewriter-like terminals. Usage is almost always:

```
eqn files | troff
neqn files | nroff
```

or equivalent.

If no files are specified (or if – is specified as the last argument), these programs read the standard input. A line beginning with **.EQ** marks the start of an equation; the end of an equation is marked by a line beginning with **.EN**. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to designate two characters as *delimiters*; subsequent text between delimiters is then treated as *eqn* input. Delimiters may be set to characters *x* and *y* with the command-line argument **–d***xy* or (more commonly) with **delim** *xy* between **.EQ** and **.EN**. The left and right delimiters may be the same character; the dollar sign is often used as such a delimiter. Delimiters are turned off by **delim off**. All text that is neither between delimiters nor between **.EQ** and **.EN** is passed through untouched.

The program *checkeq* reports missing or unbalanced delimiters and **.EQ**/**.EN** pairs.

Tokens within *eqn* are separated by spaces, tabs, new-lines, braces, double quotes, tildes, and circumflexes. Braces { } are used for grouping; generally speaking, anywhere a single character such as *x* could appear, a complicated construction enclosed in braces may be used instead. Tilde (~) represents a full space in the output, circumflex (^) half as much.

Subscripts and superscripts are produced with the keywords **sub** and **sup**. Thus *x sub j* makes x_j, *a sub k sup 2* produces $a_k{}^2$, while $e^{x^2+y^2}$ is made with *e sup {x sup 2 + y sup 2}*. Fractions are made with **over**: *a over b* yields $\frac{a}{b}$; **sqrt** makes square roots: *1 over sqrt {ax sup 2+bx+c}* results in $\frac{1}{\sqrt{ax^2+bx+c}}$.

The keywords **from** and **to** introduce lower and upper limits: $\lim_{n \to \infty} \sum_{0}^{n} x_i$ is made with *lim from {n -> inf} sum from 0 to n x sub i*. Left and right brackets, braces, etc., of the right height are made with **left** and **right**:

left [x sup 2 + y sup 2 over alpha right] ~=~ 1 produces $\left[x^2+\frac{y^2}{\alpha}\right]=1$. Legal characters after **left** and **right** are braces, brackets, bars, **c** and **f** for ceiling and floor, and **" "** for nothing at all (useful for a right-side-only bracket). A **left** *thing* need not have a matching **right** *thing*.

Vertical piles of things are made with **pile**, **lpile**, **cpile**, and **rpile**: *pile {a above b above c}* produces $\begin{matrix}a\\b\\c\end{matrix}$. Piles may have arbitrary numbers of elements; **lpile** left-justifies, **pile** and **cpile** center (but with different vertical spacing), and **rpile** right justifies. Matrices are made with **matrix**:
matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }
produces $\begin{matrix}x_i & 1\\ y_2 & 2\end{matrix}$. In addition, there is **rcol** for a right-justified column.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **tilde**, **bar**, **vec**, **dyad**, and **under**: *x dot = f(t) bar* is $\dot{x}=\overline{f(t)}$, *y dotdot bar ~=~ n under* is $\ddot{\overline{y}}=\underline{n}$, and *x vec ~=~ y dyad* is $\vec{x}=\vec{\vec{y}}$.

Point sizes and fonts can be changed with **size** *n* or **size** $\pm n$, **roman**, **italic**, **bold**, and **font** *n*. Point sizes and fonts can be changed globally in a document by **gsize** *n* and **gfont** *n*, or by the command-line arguments **-s***n* and **-f***n*.

Normally, subscripts and superscripts are reduced by 3 points from the previous size; this may be changed by the command-line argument **-p***n*.

Successive display arguments can be lined up. Place **mark** before the desired lineup point in the first equation; place **lineup** at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with **define**:

define thing % replacement %

defines a new token called *thing* that will be replaced by *replacement* whenever it appears thereafter. The % may be any character that does not occur in *replacement*.

Keywords such as **sum** ($\sum$), **int** ($\int$), **inf** (∞), and shorthands such as **>=** ($\geq$), **!=** ($\neq$), and **->** ($\rightarrow$) are recognized. Greek letters are spelled out in the desired case, as in **alpha** (α), or **GAMMA** (Γ). Mathematical words such as **sin**, **cos**, and **log** are made Roman automatically. *Troff*(1) four-character escapes such as **\(dd** (‡) and **\(bs** (⊘) may be used anywhere. Strings enclosed in double quotes (**"**...**"**) are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *troff* when all else fails. Full details are given in the manual cited below.

SEE ALSO

Typesetting Mathematics–User's Guide by B. W. Kernighan and L. L. Cherry.
cw(1), mm(1), mmt(1), nroff(1), tbl(1), eqnchar(5), mm(5).

BUGS

To embolden digits, parentheses, etc., it is necessary to quote them, as in **bold "12.3"**.

NAME

ex, edit – text editor

SYNOPSIS

ex [–] [**–v**] [**–t** tag] [**–r**] [**+***command*] [**–l**] name ...

edit [ex options]

DESCRIPTION

Ex is the root of a family of editors: *edit, ex* and *vi. Ex* is a superset of *ed,* with the most notable extension being a display editing facility. Display based editing is the focus of *vi.*

If you have not used *ed,* or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed.*

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(1), which is a command which focuses on the display editing portion of *ex.* Note that the ability to edit encrypted files is present only in the domestic (U.S.) version of the UNIX PC software.

DOCUMENTATION

The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers or the UNIX system.

The *Ex Reference Manual – Version 3.5* is a comprehensive and complete manual for the command mode features of *ex,* but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex* see the editing documents written by Brian Kernighan for the editor *ed;* the material in the introductory and advanced documents works also with *ex.*

An Introduction to Display Editing with Vi introduces the display editor *vi* and provides reference material on *vi.* All of these documents can be found in volume 2c of the Programmer's Manual. In addition, the *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction.*

FILES

/usr/lib/ex?.?strings	error messages
/usr/lib/ex?.?recover	recover command
/usr/lib/ex?.?preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), vi(1), termcap(5), environ(5)

BUGS

The *undo* command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The *z* command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line '–' option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arguments

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to the shell must be escaped. Note that **0** is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2's complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr \| *expr*
returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr \& *expr*
returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr { =, \>, \>=, \<, \<=, != } *expr*
returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr { +, – } *expr*
addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
multiplication, division, or remainder of the integer-valued arguments.

expr : *expr*
The matching operator : compares the first argument with the second argument which must be a regular expression; regular expression syntax is the same as that of *ed*(1), except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (**0** on failure). Alternatively, the \(. . . \) pattern symbols can be used to return a portion of the first argument.

EXAMPLES

1. a=`expr $a + 1`

   adds 1 to the shell variable **a**.

2. # 'For $a equal to either "/usr/abc/file" or just "file"'
   expr $a : '.*/\(.*\)' \| $a

returns the last segment of a path name (i.e., file). Watch out for / alone as an argument: *expr* will take it as the division operator (see BUGS below).

3. # A better representation of example 2.
   expr //$a : '.*/\(.*\)'

   The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. expr $VAR : '.*'

   returns the number of characters in **$VAR**.

SEE ALSO

ed(1), sh(1).

EXIT CODE

As a side effect of expression evaluation, *expr* returns the following exit values:

0	if the expression is neither null nor **0**
1	if the expression *is* null or **0**
2	for invalid expressions.

DIAGNOSTICS

syntax error	for operator/operand errors
non-numeric argument	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If **$a** is an =, the command:

expr $a = '='

looks like:

expr = = =

as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:

expr X$a = X=

NAME

factor – factor a number

SYNOPSIS

factor [number]

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2×10^{16}) it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If *factor* is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime. It takes 1 minute to factor a prime near 10^{14} on a PDP-11.

DIAGNOSTICS

"Ouch" for input out of range or for garbage input.

NAME

fc – copy floppy diskettes

SYNOPSIS

fc

DESCRIPTION

Fc makes duplicate copies of floppy diskettes, prompting for source diskette and target diskette.

NAME

file – determine file type

SYNOPSIS

file [**–c**] [**–f** ffile] [**–m** mfile] arg ...

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable **a.out**, *file* will print the version stamp, provided it is greater than 0 (see *ld*(1)).

If the **–f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

File uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of */etc/magic* explains its format.

The **–m** option instructs *file* to use an alternate magic file.

The **–c** flag causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under **–c**.

NAME

find – find files

SYNOPSIS

find path-name-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each path name in the *path-name-list* (i.e., one or more path names) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, −*n* means less than *n* and *n* means exactly *n*.

−**name** *file* True if *file* matches the current file name. Normal shell argument syntax may be used if escaped (watch out for [, ? and *).

−**perm** *onum* True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(2)) become significant and the flags are compared:

(flags&onum)==onum

−**type** *c* True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, fifo (a.k.a named pipe), or plain file.

−**links** *n* True if the file has *n* links.

−**user** *uname* True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the **/etc/passwd** file, it is taken as a user ID.

−**group** *gname* True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the **/etc/group** file, it is taken as a group ID.

−**size** *n* True if the file is *n* blocks long (512 bytes per block).

−**atime** *n* True if the file has been accessed in *n* days.

−**mtime** *n* True if the file has been modified in *n* days.

−**ctime** *n* True if the file has been changed in *n* days.

−**exec** *cmd* True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument {} is replaced by the current path name.

−**ok** *cmd* Like −**exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.

-print Always true; causes the current path name to be printed.

-cpio *device* Write the current file on *device* in *cpio* (4) format (5120 byte records).

-newer *file* True if the current file has been modified more recently than the argument *file*.

-inum*n* True if the current file is inode number *n*.

-depth Always true. Must begin the expression. Forces a *depth first* search: *find* does not apply the expression to a directory until it has applied the expression to all the files in the directory. This is useful with *cpio*; see the example in *cpio*(1). If the example were done without **-depth**, the modification dates on the copied directories would not match their originals.

(*expression*) True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

1) The negation of a primary (**!** is the unary *not* operator).

2) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).

3) Alternation of primaries (**-o** is the *or* operator).

EXAMPLE

To remove all files named **a.out** or ***.o** that have not been accessed for a week:

```
find / \( -name a.out -o -name '*.o' \) -atime +7 -exec rm {} \;
```

FILES

/etc/passwd, /etc/group

SEE ALSO

cpio(1), sh(1), test(1), stat(2), cpio(4), fs(4).

NAME

get – get a version of an SCCS file

SYNOPSIS

get [**–r**SID] [**–c**cutoff] [**–i**list] [**–x**list] [**–a**seq-no.] [**–k**] [**–e**] [**–l**[p]] [**–p**] [**–m**] [**–n**] [**–s**] [**–b**] [**–g**] [**–t**] file ...

DESCRIPTION

Get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with –. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also *FILES*, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–r*SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the **–e** keyletter is also used), as a function of the SID specified.

–c*cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **–c7502** is equivalent to **–c750228235959**. Any number of non-numeric characters may separate the various 2 digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: **"–c77/2/2 9:22:25"**. Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within the input to a command:

~!get "–c%E% %U%" s.file

–e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The **–e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents

further *get*s for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file (see *admin*(1)). Concurrent use of **get** **-e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an **-e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the **-k** keyletter in place of the **-e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see *admin*(1)) are enforced when the **-e** keyletter is used.

-b Used with the **-e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file (see *admin*(1)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)
Note: A branch *delta* may always be created from a non-leaf *delta*.

-i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID - SID

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

-x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **-i** keyletter for the *list* format.

-k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **-k** keyletter is implied by the **-e** keyletter.

-l[p] Causes a delta summary to be written into an *l-file*. If **-lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

-p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **-s** keyletter is used, in which case it disappears.

-s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text

line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

-n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **-m** and **-n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **-m** keyletter generated format.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

-t Used to access the most recently created ("top") delta in a given release (e.g., **-r1**), or release and level (e.g., **-r1.2**).

-a*seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter, and users should not use it. If both the **-r** and **-a** keyletters are specified, the **-a** keyletter is used. Care should be taken when using the **-a** keyletter in conjunction with the **-e** keyletter, as the SID of the delta to be created may not be what one expects. The **-r** keyletter can be used with the **-a** and **-e** keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **-e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **-i** keyletter is used included deltas are listed following the notation "Included"; if the **-x** keyletter is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	−**b** Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release > R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This is used to force creation of the *first* delta in a *new* release.

\# Successor.

† The **−b** keyletter is effective only if the **b** flag (see *admin* (1)) is present in the file. An entry of − means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file (see *admin*(1)), or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file (see *admin*(1)).
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file (see *admin*(1)).
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string **@(#)** recognizable by *what*(1).
%W%	A shorthand notation for constructing *what*(1) strings for UNIX program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non-UNIX program files. %A% = %Z%%Y% %M% %I%%Z%

FILES

Several auxiliary files may be created by *get*, These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **−p** keyletter is used). A *g-file* is created in all cases, whether or not any lines of text were

generated by the *get*. It is owned by the real user. If the **-k** keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **-l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
   * otherwise.
b. A blank character if the delta was applied or wasn't applied and ignored;
   * if the delta wasn't applied and wasn't ignored.
c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included.
   "X": Excluded.
   "C": Cut off (by a **-c** keyletter).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created *delta*.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an **-e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **-e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, (see *admin*(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the **-i** keyletter argument if it was present, followed by a blank and the **-x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The

same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1), sccsfile(4).
Source Code Control System in the *UNIX System Support Tools Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the **−e** keyletter is used.

NAME

getopt – parse command options

SYNOPSIS

set -- **`getopt** optstring $*`

DESCRIPTION

Getopt is used to break up options in command lines for easy parsing by shell procedures and to check for legal options. *Optstring* is a string of recognized option letters (see *getopt*(3C)); if a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. The special option -- is used to delimit the end of the options. If it is used explicitly, *getopt* will recognize it; otherwise, *getopt* will generate it; in either case, *getopt* will place it at the end of the options. The shell's positional parameters ($1 $2 . . .) are reset so that each option is preceded by a – and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an argument:

```
set -- `getopt abo: $*`
if [ $? != 0 ]
then
        echo $USAGE
        exit 2
fi
for i in $*
do
        case $i in
        -a | -b)        FLAG=$i; shift;;
        -o)             OARG=$2; shift 2;;
        --)             shift; break;;
        esac
done
```

This code will accept any of the following as equivalent:

```
cmd -aoarg file file
cmd -a -o arg file file
cmd -oarg -a file file
cmd -a -oarg -- file file
```

SEE ALSO

sh(1), getopt(3C).

DIAGNOSTICS

Getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

NAME

greek – select terminal filter

SYNOPSIS

greek [**–T**terminal]

DESCRIPTION

Greek is a filter that reinterprets the extended character set, as well as the reverse and half-line motions, of a 128-character TELETYPE Model 37 terminal (which is the *nroff* default terminal) for certain other terminals. Special characters are simulated by overstriking, if necessary and possible. If the argument is omitted, *greek* attempts to use the environment variable **$TERM** (see *environ*(5)). The following *terminal*s are recognized currently:

300	DASI 300.
300-12	DASI 300 in 12-pitch.
300s	DASI 300s.
300s-12	DASI 300s in 12-pitch.
450	DASI 450.
450-12	DASI 450 in 12-pitch.
1620	Diablo 1620 (alias DASI 450).
1620-12	Diablo 1620 (alias DASI 450) in 12-pitch.
2621	Hewlett-Packard 2621, 2640, and 2645.
2640	Hewlett-Packard 2621, 2640, and 2645.
2645	Hewlett-Packard 2621, 2640, and 2645.
4014	Tektronix 4014.
hp	Hewlett-Packard 2621, 2640, and 2645.
tek	Tektronix 4014.

FILES

/usr/bin/300
/usr/bin/300s
/usr/bin/4014
/usr/bin/450
/usr/bin/hp

SEE ALSO

300(1), 4014(1), 450(1), eqn(1), mm(1), nroff(1), environ(5), greek(5), term(5).

NAME

grep, egrep, fgrep – search a file for a pattern

SYNOPSIS

grep [options] expression [files]
egrep [options] [expression] [files]
fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *grep* family search 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 *ed*(1); *grep* uses a compact non-deterministic algorithm. *Egrep* patterns are full regular *expressions*; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed *strings*; it is fast and compact. The following *options* are recognized:

–v All lines but those matching are printed.
–c Only a count of matching lines is printed.
–i Ignore upper/lower case distinction during compare.
–x (Exact) only lines matched in their entirety are printed (fgrep only).
–l Only the names of files with matching lines are listed (once), separated by new-lines.
–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.
–s The error messages produced for nonexistent or unreadable files are suppressed (*grep* only).
–e*expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a – (does not work with *grep*).
–f*file* The regular *expression* (*egrep*) or *strings* list (*fgrep*) is taken from the *file*.

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

Fgrep searches for lines that contain one of the *strings* separated by new-lines.

Egrep accepts regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a new-line match strings that are matched by either.
4. A regular expression may be enclosed in parentheses () for grouping

The order of precedence of operators is [ ], then * ? +, then concatenation, then | and new-line.

SEE ALSO

ed(1), sed(1), sh(1).

DIAGNOSTICS

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

BUGS

Ideally there should be only one *grep*, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to 256 characters; longer lines are truncated. *Egrep* does not recognize ranges, such as **[a–z]**, in character classes.

When using the **–b** option, *grep* reports the block number of the match in 512-byte blocks; *egrep* and *fgrep* report the block number in 1024-byte blocks.

NAME

head – give first few lines

SYNOPSIS

head [– *count*] [*file* ...]

DESCRIPTION

Head gives the first *count* lines of each of the specified files. If no files are specified, *head* reads the standard input. If you omit *count, head* prints the first 10 lines.

SEE ALSO

tail(1).

NAME

help – ask for help

SYNOPSIS

help [args]

DESCRIPTION

Help finds information to explain a message from a command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1	Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., **ge6**, for message 6 from the *get* command).
type 2	Does not contain numerics (as a command, such as **get**)
type 3	Is all numeric (e.g., **212**)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try "help stuck".

FILES

/usr/lib/help	directory containing files of message text.
/usr/lib/help/helploc	file containing locations of help files not in **/usr/lib/help**.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

hp – handle special functions of HP 2640 and 2621-series terminals

SYNOPSIS

hp [**–e**] [**–m**]

DESCRIPTION

Hp supports special functions of the Hewlett-Packard 2640 series of terminals, with the primary purpose of producing accurate representations of most *nroff* output. A typical use is:

nroff –h files . . . | hp

Regardless of the hardware options on your terminal, *hp* tries to do sensible things with underlining and reverse line-feeds. If the terminal has the "display enhancements" feature, subscripts and superscripts can be indicated in distinct ways. If it has the "mathematical-symbol" feature, Greek and other special characters can be displayed.

The flags are as follows:

–e It is assumed that your terminal has the "display enhancements" feature, and so maximal use is made of the added display modes. Overstruck characters are presented in the Underline mode. Superscripts are shown in Half-bright mode, and subscripts in Half-bright, Underlined mode. If this flag is omitted, *hp* assumes that your terminal lacks the "display enhancements" feature. In this case, all overstruck characters, subscripts, and superscripts are displayed in Inverse Video mode, i.e., dark-on-light, rather than the usual light-on-dark.

–m Requests minimization of output by removal of new-lines. Any contiguous sequence of 3 or more new-lines is converted into a sequence of only 2 new-lines; i.e., any number of successive blank lines produces only a single blank output line. This allows you to retain more actual text on the screen.

With regard to Greek and other special characters, *hp* provides the same set as does *300*(1), except that "not" is approximated by a right arrow, and only the top half of the integral sign is shown. The display is adequate for examining output from *neqn*.

DIAGNOSTICS

"line too long" if the representation of a line exceeds 1,024 characters.

The exit codes are **0** for normal termination, **2** for all errors.

SEE ALSO

300(1), col(1), eqn(1), greek(1), nroff(1), tbl(1).

BUGS

An "overstriking sequence" is defined as a printing character followed by a backspace followed by another printing character. In such sequences, if either printing character is an underscore, the other printing character is shown underlined or in Inverse Video; otherwise, only the first printing character is shown (again, underlined or in Inverse Video). Nothing special is done if a backspace

is adjacent to an ASCII control character. Sequences of control characters (e.g., reverse line-feeds, backspaces) can make text "disappear"; in particular, tables generated by *tbl*(1) that contain vertical lines will often be missing the lines of text that contain the "foot" of a vertical line, unless the input to *hp* is piped through *col*(1).
Although some terminals do provide numerical superscript characters, no attempt is made to display them.

NAME

hyphen – find hyphenated words

SYNOPSIS

hyphen [files]

DESCRIPTION

Hyphen finds all the hyphenated words ending lines in *files* and prints them on the standard output. If no arguments are given, the standard input is used; thus, *hyphen* may be used as a filter.

EXAMPLE

The following will allow the proofreading of *nroff*'s hyphenation in *textfile*.

```
mm textfile | hyphen
```

SEE ALSO

mm(1).

BUGS

Hyphen can't cope with hyphenated *italic* (i.e., underlined) words; it will often miss them completely, or mangle them.

Hyphen occasionally gets confused, but with no ill effects other than spurious extra output.

NAME

id – print user and group IDs and names

SYNOPSIS

id

DESCRIPTION

Id writes a message on the standard output giving the user and group IDs and the corresponding names of the invoking process. If the effective and real IDs do not match, both are printed.

SEE ALSO

logname(1), getuid(2).

NAME

ipcrm – remove a message queue, semaphore set or shared memory id

SYNOPSIS

ipcrm [*options*]

DESCRIPTION

Ipcrm will remove one or more specified message, semaphore or shared memory identifiers. The identifiers are specified by the following *options*:

–q *msqid* removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.

–m *shmid* removes the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

–s *semid* removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.

–Q *msgkey* removes the message queue identifier, created with key *msgkey*, from the system and destroys the message queue and data structure associated with it.

–M *shmkey* removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

–S *semkey* removes the semaphore identifier, created with key *semkey*, from the system and destroys the set of semaphores and data structure associated with it.

The details of the removes are described in *msgctl*(2), *shmctl*(2), and *semctl*(2). The identifiers and keys may be found by using *ipcs*(1).

SEE ALSO

ipcs(1), msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2).

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

ipcs [options]

DESCRIPTION

Ipcs prints certain information about active inter-process communication facilities. Without *options*, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following *options*:

−q Print information about active message queues.

−m Print information about active shared memory segments.

−s Print information about active semaphores.

If any of the options **−q**, **−m**, or **−s** are specified, information about only those indicated will be printed. If none of these three are specified, information about all three will be printed.

−b Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.) See below for meaning of columns in a listing.

−c Print creator's login name and group name. See below.

−o Print information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments.)

−p Print process number information. (Process ID of last process to send a message and process ID of last process to receive a message on message queues and process ID of creating process and process ID of last process to attach or detach on shared memory segments) See below.

−t Print time information. (Time of the last control operation that changed the access permissions for all facilities. Time of last *msgsnd* and last *msgrcv* on message queues, last *shmat* and last *shmdt* on shared memory, last *semop*(2) on semaphores.) See below.

−a Use all print *options*. (This is a shorthand notation for **−b**, **−c**, **−o**, **−p**, and **−t**.)

−C *corefile* Use the file *corefile* in place of **/dev/kmem**.

−N *namelist* The argument will be taken as the name of an alternate *namelist* (**/unix** is the default).

The column headings and the meaning of the columns in an *ipcs* listing are given below; the letters in parentheses indicate the *options* that cause the corresponding heading to appear; **all** means that the heading always appears. Note that these *options* only determine what information is provided for each facility; they do *not* determine which facilities will be listed.

T (all) Type of the facility:
- **q** message queue;
- **m** shared memory segment;
- **s** semaphore.

ID (all) The identifier for the facility entry.

KEY (all) The key used as an argument to *msgget*, *semget*, or *shmget* to create the facility entry. (Note: The key of a shared memory segment is changed to **IPC_PRIVATE** when the segment has been removed until all processes attached to the segment detach it.)

MODE (all) The facility access modes and flags: The mode consists of 11 characters that are interpreted as follows:

The first two characters are:
- **R** if a process is waiting on a *msgrcv*;
- **S** if a process is waiting on a *msgsnd*;
- **D** if the associated shared memory segment has been removed. It will disappear when the last process attached to the segment detaches it;
- **C** if the associated shared memory segment is to be cleared when the first attach is executed;
- – if the corresponding special flag is not set.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused.

The permissions are indicated as follows:
- **r** if read permission is granted;
- **w** if write permission is granted;
- **a** if alter permission is granted;
- – if the indicated permission is *not* granted.

OWNER	(all)	The login name of the owner of the facility entry.
GROUP	(all)	The group name of the group of the owner of the facility entry.
CREATOR	(a,c)	The login name of the creator of the facility entry.
CGROUP	(a,c)	The group name of the group of the creator of the facility entry.
CBYTES	(a,o)	The number of bytes in messages currently outstanding on the associated message queue.
QNUM	(a,o)	The number of messages currently outstanding on the associated message queue.
QBYTES	(a,b)	The maximum number of bytes allowed in messages outstanding on the associated message queue.
LSPID	(a,p)	The process ID of the last process to send a message to the associated queue.
LRPID	(a,p)	The process ID of the last process to receive a message from the associated queue.
STIME	(a,t)	The time the last message was sent to the associated queue.
RTIME	(a,t)	The time the last message was received from the associated queue.
CTIME	(a,t)	The time when the associated entry was created or changed.
NATTCH	(a,o)	The number of processes attached to the associated shared memory segment.
SEGSZ	(a,b)	The size of the associated shared memory segment.
CPID	(a,p)	The process ID of the creator of the shared memory entry.
LPID	(a,p)	The process ID of the last process to attach or detach the shared memory segment.
ATIME	(a,t)	The time the last attach was completed to the associated shared memory segment.
DTIME	(a,t)	The time the last detach was completed on the associated shared memory segment.
NSEMS	(a,b)	The number of semaphores in the set associated with the semaphore entry.
OTIME	(a,t)	The time the last semaphore operation was completed on the set associated with the semaphore entry.

FILES

/unix	system namelist
/dev/kmem	memory
/etc/passwd	user names
/etc/group	group names

SEE ALSO

msgop(2), semop(2), shmop(2), stdipc(3C).

BUGS

Things can change while *ipcs* is running; the picture it gives is only a close approximation to reality.

NAME

join – relational database operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is –, the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

Fields are normally separated by blank, tab or new-line. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

–a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s* Replace empty output fields by string *s*.

–j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.

–o *list* Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number.

–t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

awk(1), comm(1), sort(1).

BUGS

With default field separation, the collating sequence is that of **sort –b**; with **–t**, the sequence is that of a plain sort.

The conventions of *join*, *sort*, *comm*, *uniq* and *awk*(1) are wildly incongruous.

NAME

kill – terminate a process

SYNOPSIS

kill [–signo] PID ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with & is reported by the Shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using *ps*(1).

The details of the kill are described in *kill*(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The killed process must belong to the current user unless he is the super-user.

If a signal number preceded by – is given as first argument, that signal is sent instead of terminate (see *signal*(2)). In particular "kill –9 . . ." is a sure kill.

SEE ALSO

ps(1), sh(1), kill(2), signal(2).

NAME

ksh – Korn shell command programming language

SYNOPSIS

ksh [**-acefhikmnorstuvx**] [**-o** option ...] [arg ...]

DESCRIPTION

Ksh is a command programming language that executes commands read from a terminal or a file. See *Invocation* below for the meaning of arguments to the Korn shell.

Definitions.

A *metacharacter* is one of the following characters:

; & ( ) | < > new-line space tab

A *blank* is a **tab** or a **space**. An *identifier* is a sequence of letters, digits, or underscores starting with a letter or underscore. Identifiers are used as names for *aliases*, *functions*, and *named parameters*. A *word* is a sequence of *characters* separated by one or more non-quoted *metacharacters*.

Commands.

A *simple-command* is a sequence of *blank* separated words which may be preceded by a parameter assignment list (see *Environment* below). 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 (octal) 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 Korn shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **&&**, or ||, and optionally terminated by **;**, **&**, or **|&**. Of these five symbols, **;**, **&**, and **|7** have equal precedence, which is lower than that of **&&** and ||. The symbols **&&** and || also have equal precedence. A semicolon (**;**) causes sequential execution of the preceding pipeline; an ampersand (**&**) causes asynchronous execution of the preceding pipeline (i.e., the Korn shell does *not* wait for that pipeline to finish). The symbol **|&** causes asynchronous execution of the preceding command or pipeline with a two-way pipe established to the parent shell. The standard input and output of the spawned command can be written to and read from by the parent Shell using the **-p** option of the special commands **read** and **print** described later. Only one such command can be active at any given time. The symbol **&&** (||) causes the *list* following it to be executed only if the preceding pipeline returns a zero value. An arbitrary number of new-lines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a simple-command or one of the following. Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

for *identifier* [**in** *word* **...**] **do** *list* **done**
Each time a **for** command is executed, *identifier* is set to the next *word* taken from the **in** *word* list. If **in** *word* **...** is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see *Parameter Substitution* below). Execution ends when there are no more words in the list.

select *identifier* [**in** *word* **...**] **do** *list* **done**
A **select** command prints on standard error (file descriptor 2), the set of *words*, each preceded by a number. If **in** *word* **...** is omitted, then the positional parameters are used instead (see *Parameter Substitution* below). The **PS3** prompt is printed and a line is read from the standard input. If this line consists of the number of one of the listed **words**, then the value of the parameter *identifier* is set to the *word* corresponding to this number. If this line is empty the selection list is printed again. Otherwise the value of the parameter *identifier* is set to **null**. The contents of the line read from standard input is saved in the parameter **REPLY**. The *list* is executed for each selection until a **break** or *end-of-file* is encountered.

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 usef for file-name generation (see *File Name Generation* below).

if *list* **then** *list* [**elif** *list* **then** *list*] **...** [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns a zero exit status.

while *list* **do** *list* **done**
until *list* **do** *list* **done**
A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

(*list*) Execute *list* in a separate environment. Note that, if two adjacent open parentheses are needed for nesting, a space must be inserted to avoid arithmetic evaluation as described below.

{ *list* ; }
List is simply executed. Note that { is a *keyword* and requires a blank in order to be recognized.

function *identifier* { *list* ; }
identifier ( ) { *list* ; }
Define a function which is referenced by *identifier*. The body of the function is the *list* of commands between { and } (see *Functions* below).

time *pipeline*
The *pipeline* is executed and the elapsed time as well as the user and system time are printed on standard error.

The following keywords are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { } function select time

Comments.
A word beginning with # causes that word and all the following characters up to a new-line to be ignored.

Aliasing.
The first word of each command is replaced by the text of an **alias** if an **alias** for this word has been defined. The first character of an **alias** name can be any printable character, but the rest of the characters must be the same as for a valid *identifier*. The replacement string can contain any valid Korn Shell script including the metacharacters listed above. The first word of each command of the replaced text will not be tested for additional aliases. Aliases can be used to redefine special built-in commands but cannot be used to redefine the keywords listed above. Aliases can be created, listed, and exported with the **alias** command and can be removed with the **unalias** command. Exported aliases remain in effect for sub-shells but must be reinitialized for separate invocations of the Korn Shell (see *Invocation* below).

Aliasing is performed when scripts are read, not while they are executed. Therefore, for an alias to take effect the **alias** command has to be executed before the command which references the alias is read.

Aliases are frequently used as a shorthand for full path names. An option to the aliasing facility allows the value of the alias to be automatically set to the full pathname of the corresponding command. These aliases are called *tracked* aliases. The value of a *tracked* alias is defined the first time the identifier is read and becomes undefined each time the **PATH** variable is reset. These aliases remain *tracked* so that the next subsequent reference will redefine the value. Several tracked aliases are compiled into the shell. The **−h** option of the **set** command makes each command name which is an *identifier* into a tracked alias.

The following *exported aliases* are compiled into the shell but can be unset or redefined:

```
echo='print -'
false='let 0'
history='fc -l'
integer='typeset -i'
pwd='print - $PWD'
r='fc -e -'
true='let 1'
type='whence -v'
hash='alias -t'
```

Tilde Substitution.

After alias substitution is performed, each word is checked to see if it begins with an unquoted ~. If it does, then the word up to a / is checked to see if it matches a user name in the **/etc/passwd** file. If a match is found, the ~ and the matched login name are replaced by the login directory of the matched user. This is called a *tilde* substitution. If no match is found, the original text is left unchanged. A ~ by itself, or in front of a /, is replaced by the value of the **HOME** parameter. A ~ followed by a + or − is replaced by the value of the parameter **PWD** and **OLDPWD** respectively.

In addition, the value of each *keyword parameter* is checked to see if it begins with a ~ or if a ~ appears after a :. In either of these cases a *tilde* substitution is attempted.

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. The command substitution **`cat file`** can be replaced by the equivalent but faster **`<file`**. Command substitution of special commands that do not perform input/output redirection are carried out without creating a separate process.

Parameter Substitution.

A *parameter* is an *identifier*, a digit, or any of the characters *, @, #, ?, −, $, and !. A *named parameter* (a parameter denoted by an identifier) has a *value* and zero or more *attributes*. *Named parameters* can be assigned *values* and *attributes* by using the **typeset** special command. The attributes supported by the Korn Shell are described later with the **typeset** special command. Exported parameters pass values and attributes to sub-shells but only values to the environment.

The Korn Shell supports a limited one-dimensional array facility. An element of an array parameter is referenced by a *subscript*. A *subscript* is denoted by a [, followed by an *arithmetic expression* (see *Arithmetic Evaluation* below) followed by a]. The value of all subscripts must be in the range of 0 through 127. Arrays need not be declared. Any reference to a named parameter with a valid subscript is legal and an array will be created if necessary. Referencing an array without a subscript is equivalent to referencing the first element.

The *value* of a *named parameter* may also be assigned by writing:

name=*value* [*name*=*value*] ...

If the integer attribute, **-i**, is set for *name*, the *value* is subject to arithmetic evaluation as described below. Positional parameters, parameters denoted by a number, may be assigned values with the **set** special command. Parameter **$0** is set from argument zero when the Korn Shell is invoked. The character $ is used to introduce substitutable *parameters*.

${*parameter*}
The value, if any, of the parameter is substituted. The braces are required when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name or when a named parameter is subscripted. 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). If an array *identifier* with subscript * or @ is used, then the value for each of the elements is substituted (separated by spaces).

${#*parameter*}
If *parameter* is not *, the length of the value of the *parameter* is substituted. Otherwise, the number of positional parameters is substituted.

${#*identifier*[*]}
The number of elements in the array *identifier* is substituted.

${*parameter*:-*word*}
If *parameter* is set and non-null then substitute its value; otherwise substitute *word*.

${*parameter*:=*word*}
If *parameter* is not set or is null 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 and is non-null 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 and is non-null then substitute *word*; otherwise substitute nothing.

${*parameter*#*pattern*}
${*parameter*##*pattern*}
If the Korn Shell *pattern* matches the beginning of the value of *parameter*, then the value of this substitution is the value of the *parameter* with the matched portion deleted; otherwise the value of this *parameter* is substituted.

${*parameter*%*pattern*}
${*parameter*%%*pattern*}
If the Korn Shell *pattern* matches the end of the value of *parameter*, then substitute the value of *parameter* with the matched part deleted; otherwise substitute the value of *parameter*.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is null:

```
echo ${d:-`pwd`}
```

If the colon (:) is omitted from the above expressions, then the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the Korn Shell:

The number of positional parameters in decimal.

- Flags supplied to the Korn Shell on invocation or by the **set** command.

? The decimal value returned by the last executed command.

$ The process number of this shell.

_ The last argument of the previous command.

! The process number of the last background command invoked.

PPID The process number of the parent of the shell.

PWD The present working directory set by the **cd** command.

RANDOM
Each time this parameter is referenced, a random integer is generated. The sequence of random numbers can be initialized by assigning a numeric value to **RANDOM**.

REPLY
This parameter is set by the **select** statement and by the **read** special command when no arguments are supplied.

The following parameters are used by the Korn shell:

CDPATH
the search path for the **cd** command.

COLUMNS
If this variable is set, the value is used to define the width of the edit window for the shell edit modes and for printing **select** lists.

EDITOR
If the value of either of these variables ends in

emacs, *gmacs*, or *vi*, then the corresponding option (see Special Command **set** below) will be turned on.

ENV If this parameter is set, then command and parameter substitution is performed on the value to generate the pathname of the script that will be executed when the Korn shell is invoked (see *Invocation* below). This file is typically used for *alias* and *function* definitions.

FCEDIT
The default editor name for the **fc** command.

IFS Internal field separators, normally **space**, **tab**, and **new-line**, that are used to separate command words which result from command or parameter substitution and for separating words with the special command **read**.

HISTFILE
If this parameter is set when the Korn shell is invoked, then the value is the pathname of the file that will be used to store the command history (see *Command Re-entry* below).

HISTSIZE
If this parameter is set when the Korn shell is invoked, then the number of previously entered commands that are accessible by this shell will be greater than or equal to this number. The default is 128.

HOME
The default argument (home directory) for the **cd** command.

MAIL If this parameter is set to the name of a mail file *and* the **MAILPATH** parameter is not set, then the shell informs the user of arrival of mail in the specified file.

MAILCHECK
This variable specifies how often (in seconds) the shell will check for changes in the modification time of any of the files specified by the **MAILPATH** or **MAIL** parameters. The default value is 600 seconds. If set to 0, the shell will check before each prompt.

MAILPATH
A colon (**:**) separated list of file names. If this parameter is set then the shell informs the user of any modifications to the specified files that have occurred within the last **MAILCHECK** seconds. Each file name can be followed by a **?** and a message that will be printed. The message will undergo parameter and command substitution

with the parameter $_ defined as the name of the file that has changed. The default message is *you have mail in* $_."

PATH
The search path for commands (see *Execution* below).

PS1 The value of this parameter is expanded for parameter substitution to define the primary prompt string, which by default is "$". The character ! in a prompt string is replaced by the command number (see *Command Re-entry* below).

PS2 Secondary prompt string, by default ">".

PS3 Selection prompt string used within a **select** loop, by default "**#?**".

SHELL
The pathname of the *shell* is kept in the environment. At invocation, if the value of this variable contains an **r** in the basename, then the shell becomes restricted.

TMOUT
If set to a value greater than zero, the shell will terminate if a command is not entered within the prescribed number of seconds. (Note that the shell can be compiled with a maximum bound for this value which cannot be exceeded.)

VISUAL
If the value of this variable ends in *emacs*, *gmacs*, or *vi*, then the corresponding option (see Special Command **set** below) will be turned on.

The shell gives default values to **PATH**, **PS1**, **PS2**, **MAILCHECK**, **TMOUT**, and **IFS**, while **HOME**, **ENV**, **SHELL**, and **MAIL** are not set at all by the shell (although **HOME** *is* set by *login*(1M)). On some systems **MAIL** and **SHELL** are also set by *login*(1M)).

Blank Interpretation.
After parameter and command substitution, the results of substitution are scanned for the 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 [unless the **-f** option has been **set**. 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. When a *pattern* is used for file name generation, the character . at the start of a file name or

immediately following a /, as well as the character / itself, must be matched explicitly. In other instances of pattern matching the / and . are not treated specially.

*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening "[" is a "! " then any character not enclosed is matched. A – can be included in the character set by putting it as the first or last character.

Quoting.

Each of the *metacharacters* listed above (see *Definitions* above) has a special meaning to the Korn shell and causes termination of a word unless quoted. A character may be *quoted* (i.e., made to stand for itself) by preceding it with a \. The pair **\new-line** is ignored. All characters enclosed between a pair of single quote marks (''), except a single quote, are quoted. Inside double quote marks (""), parameter and command substitution occurs and \ quotes the characters \, ', ", and $. "$*" is equivalent to "$1 $2 ... ", whereas "$@" is equivalent to "$1" "$2" ....

The special meaning of keywords can be removed by quoting any character of the keyword. The recognition of special command names listed below cannot be altered by quoting them.

Arithmetic Evaluation.

An ability to perform integer arithmetic is provided with the special command **let**. Evaluations are performed using *long* arithmetic. Constants are of the form [*base*#]*n* where *base* is a decimal number between 2 and 36 representing the arithmetic base and *n* is a number in that base. If *base* is omitted then base 10 is used.

An internal integer representation of a *named parameter* can be specified with the **–i** option of the **typeset** special command. When this attribute is selected the first assignment to the parameter determines the arithmetic base to be used when parameter substitution occurs.

Since many of the arithmetic operators require quoting, an alternative form of the **let** command is provided. For any command which begins with a **((**, all the characters until a matching **))** are treated as a quoted expression. More precisely, **((...))** is equivalent to **let " ... "**.

Prompting.

When used interactively, the shell prompts with the value of **PS1** before reading a command. If at any time a new-line is typed and further input is needed to complete a command, then the secondary prompt (i.e., the value of **PS2**) is issued.

Input/Output.

Before a command is executed, its input and output may be

redirected using a special notation interpreted by the Korn 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. Command and parameter substitution occurs before *word* or *digit* is used except as noted below. File name generation occurs only if the pattern matches a single file and blank interpretation is not performed.

<*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 to it (by first seeking to the end-of-file); otherwise, the file is created.

<<[-]*word*
The shell input is read up to a line that is the same as *word*, or to an end-of-file. No parameter substitution, command substitution, or file name generation is performed on *word*. The resulting document, called a *here-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, **\new-line** is ignored, and \ must be used to quote the characters \, $, `, and the first character of *word*. If - is appended to <<, then all leading tabs are stripped from *word* and from the document.

<&*digit* The standard input is duplicated from file descriptor *digit* (see *dup*(2)). Similarly for the standard output using >&*digit*.

<&- 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 number referred to is that specified by the digit (instead of the default 0 or 1). For example:

... 2>&1

means file descriptor 2 is to be opened for writing as a duplicate of file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*file descriptor*, *file*) association at the time of evaluation. For example:

... 1>*fname* 2>&1

first associates file descriptor 1 with file *fname*. It then associates file descriptor 2 with the file associated with file descriptor 1 (i.e., *fname*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1

had been) and then file descriptor 1 would be associated with file *fname*.

If a command is followed by & and job control is not active, 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* (see *environ*(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The names must be *identifiers* and the values are character strings. The Korn shell interacts with the environment in several ways. On invocation, the Korn shell scans the environment and creates a parameter for each name found, giving it the corresponding value and marking it *export*. Executed commands inherit the environment. If the user modifies the values of these parameters or creates new ones, using the **export** or **typeset** **-x** commands, they become part of the environment. The environment seen by any executed command is thus composed of any name-value pairs originally inherited by the Korn shell, whose values may be modified by the current shell, plus any additions which must be noted in **export** or **typeset** **-x** commands.

The environment for any *simple-command* or function may be augmented by prefixing it with one or more parameter assignments. A parameter assignment argument is a word of the form *identifier*=*value*. Thus:

```
TERM=450 cmd args
```

and

```
(export TERM; TERM=450; cmd args)
```

are equivalent (as far as the above execution of *cmd* is concerned).

If the **-k** flag is set, *all* parameter assignment arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and then **c**:

```
echo a=b c
set -k
echo a=b c
```

Functions.

The **function** keyword, described in the *Commands* section above, is used to define shell functions. Shell functions are read in and stored internally. Alias names are resolved when the function is read. Functions are executed like commands with the arguments passed as positional parameters (see *Execution* below). Functions execute in the same process as the caller and share all files, traps (other than **EXIT** and **ERR**), and present working directory with the caller. A trap set on **EXIT** inside a function is executed after the function completes. Ordinarily, variables are shared between the calling program and the function. However, the **typeset** special command used within a function defines local

variables whose scope includes the current function and all functions it calls.

The special command **return** is used to return from function calls. Errors within functions return control to the caller.

Function identifiers can be listed with the **−f** option of the **typeset** special command. The text of functions will also be listed. Functions can be undefined with the **−f** option of the **unset** special command.

Ordinarily, functions are unset when the shell executes a shell script. The **−xf** option of the **typeset** command allows a function to be exported to scripts that are executed without a separate invocation of the shell. Functions that need to be defined across separate invocations of the shell should be placed in the **ENV** file.

Jobs.

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

[1] 1234

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

This paragraph and the next require features that are not in all versions of UNIX and may not apply. If you are running a job and wish to do something else you may press the key combination ^Z (control-Z) which sends a STOP signal to the current job. The shell will then normally indicate that the job has been "Stopped," and print another prompt. You can then manipulate the state of this job, putting it in the background with the **bg** command, or run some other commands and then eventually bring the job back into the foreground with the foreground command **fg**. A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "stty tostop." If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the Korn shell. The character **%** introduces a job name. If you wish to refer to job number 1, you can name it as **%1**. Jobs can also be named by prefixes of the string typed in to **kill** or restart them. Thus, on systems that support job control, **`fg%ed´** would normally restart a suspended *ed*(1) job, if there were a suspended job whose name began with the string 'ed'.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a + and the previous job with a −. The abbreviation %+ refers to the current job and %− refers to the previous job. %% is also a synonym for the current job.

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work.

When you try to leave the shell while jobs are running or stopped, you will be warned that 'You have stopped(running) jobs.' You may use the **jobs** command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the stopped jobs will be terminated.

Signals.

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by & and the job **monitor** option is not active. Otherwise, signals have the values inherited by the shell from its parent, with the exception of signal 11 (but see also the **trap** command below).

Execution.

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the *Special Commands* listed below, it is executed within the current shell process. Next, the command name is checked to see if it matches one of the user defined functions. If it does, the positional parameters are saved and then reset to the arguments of the *function* call. When the *function* completes or issues a **return**, the positional parameter list is restored and any trap set on **EXIT** within the function is executed. The value of a *function* is the value of the last command executed. A function is also executed in the current shell process. If a command name is not a *special command* or a user defined *function*, a process is created and an attempt is made to execute the command via *exec*(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is **:/bin:/usr/bin** (specifying the current directory, **/bin**, and **/usr/bin**, in that order). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign, between colon delimiters, or at the end of the path list. 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 a directory or an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. All non-exported aliases, functions, and named parameters are removed in this case. A parenthesized command is also executed in a sub-shell.

Command Re-entry.

The text of the last **HISTSIZE** (default 128) commands entered from a terminal device is saved in a *history* file. The file **$HOME/.history** is used if the **HISTFILE** variable is not set or is not writable. A shell can access the commands of all *interactive* shells which use the same named **HISTFILE**. The special command **fc** is used to list or edit a portion of this file. The portion of the file to be edited or listed can be selected by number or by giving the first character or characters of the command. A single command or range of commands can be specified. If you do not specify an editor program as an argument to **fc** then the value of the parameter **FCEDIT** is used. If **FCEDIT** is not defined then */bin/ed* is used. The edited command(s) are printed and re-executed upon leaving the editor. The editor name – is used to skip the editing phase and to re-execute the command. In this case a substitution parameter of the form *old=new* can be used to modify the command before execution. For example, if **r** is aliased to '**fc –e –**' then typing '**r bad=good c**' will re-execute the most recent command which starts with the letter **c**, replacing the string **bad** with the string **good**.

In-line Editing Options.

Normally, each command line entered from a terminal device is simply typed followed by a new-line ('RETURN' or 'LINE FEED'). If any of the options *emacs*, *gmacs*, or *vi* is active, the user can edit the command line. To be in any one of these edit modes **set** the corresponding option. An editing option is automatically selected each time the **VISUAL** or **EDITOR** variable is assigned a value ending in one of these option names.

The editing features require that the user's terminal accept 'RETURN' as carriage return without line feed and that a space (' ') must overwrite the current character on the screen. ADM terminal users should set the "space–advance" switch to 'space'. Hewlett-Packard series 2621 terminal users should set the straps to 'bcGHxZ etX'.

The editing modes implement a concept where the user is looking through a window at the current line. The window width is the value of **COLUMNS** if it is defined, otherwise 80. If the line is longer than the window width minus two, a mark is displayed at the end of the window to notify the user. As the cursor moves and reaches the window boundaries the window will be centered about the cursor. The mark is a > (<, *) if the line extends on the right (left, both) side(s) of the window.

Emacs Editing Mode.

This mode is entered by enabling either the *emacs* or *gmacs* option. The only difference between these two modes is the way they handle ^**T**. To edit, the user moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. All the editing commands are control characters or escape sequences. The notation for control characters is caret (^) followed by the character. For example, ^**F** is the notation for control–**F**. This is entered by pressing 'f' while holding down the

'CTRL' (control) key. The 'SHIFT' key is *not* pressed. (The notation **^?** indicates the DEL (delete) key.)

The notation for escape sequences is **M–** followed by a character. For example, **M–f** (pronounced Meta f) is entered by pressing ESC (ASCII **033**) followed by 'f'. (**M–F** would be the notation for ESC followed by 'SHIFT' (capital) 'F'.)

All edit commands operate from any place on the line (not just at the beginning). Neither the "RETURN" nor the "LINE FEED" key is entered after edit commands except when noted.

^F Move cursor forward (right) one character.

M–f Move cursor forward one word. (The editor's idea of a word is a string of characters consisting of only letters, digits and underscores.)

^B Move cursor backward (left) one character.

M–b Move cursor backward one word.

^A Move cursor to start of line.

^E Move cursor to end of line.

^]*char* Move cursor to character *char* on current line.

^X^X Interchange the cursor and mark.

erase (User-defined erase character as defined by the *stty* command, usually **^H** or **#**.) Delete previous character.

^D Delete current character.

M–d Delete current word.

M–^H (Meta-backspace) Delete previous word.

M–h Delete previous word.

M–^? (Meta-DEL) Delete previous word (if your interrupt character is **^?** (DEL, the default) then this command will not work).

^T Transpose current character with next character in *emacs* mode. Transpose two previous characters in *gmacs* mode.

^C Capitalize current character.

M–C Capitalize current word.

^K Kill from the cursor to the end of the line. If given a parameter of zero then kill from the start of line to the cursor.

^W Kill from the cursor to the mark.

M–p Push the region from the cursor to the mark on the stack.

kill (User-defined kill character as defined by the *stty* command, usually **^G** or **@**.) Kill the entire current line. If two *kill* characters are entered in succession, all kill characters from then on cause a line feed (useful when

using paper terminals).

^Y Restore last item removed from line (Yank item back to the line.)

^L Line feed and print current line.

^@ (Null character) Set mark.

M– (Meta space) Set mark.

^J (New line) Execute the current line.

^M (Return) Execute the current line.

eof End-of-file character, normally ^D, will terminate the shell if the current line is null.

^P Fetch the previous command. Each time ^P is entered the previous command back in time is accessed.

M–< Fetch the lest recent (oldest) history line.

M–> Fetch the most recent (youngest) history line.

^N Fetch next command. Each time ^N is entered the next command forward in time is accessed.

^R*string* Reverse search history for a previous command line containing *string*. If a parameter of zero if given the search is forward. *String* is terminated by a "RETURN" or "NEW LINE".

^O Operate–Execute the current line and fetch the next line relative to the current line from the history file.

M–*digits* (Escape) Define numeric parameter, the digits are taken as a parameter to the next command. The commands that accept a parameter are ^F, ^B, *erase*, ^D, ^K, ^R, ^P, and ^N.

M–*letter* Soft-key–Your alias list is searched for an alias by the name _ *letter*. If an alias of this name is defined, its value will be inserted on the line. The *letter* must not be one of the above meta-functions.

M–_ The last parameter of the previous command is inserted on the line.

M–. The last parameter of the previous command is inserted on the line.

M–* Attempt file name generation on the current word.

^U Multiply parameter of next command by 4.

\ Escape next character. Editing characters, the user's erase, kill and interrupt (normally ^?) characters may be entered in a command line or in a search string if preceded by a \. The \ removes the next character's editing features (if any).

^V Display version of the shell.

Vi **Editing Mode.**

There are two typing modes. Initially, when you enter a

command you are in the *input* mode. To edit, the user enters control mode by typing ESC (**033**) and moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. Most control commands accept an optional repeat *count* prior to the command.

When in *vi* mode on most systems, canonical processing is initially enabled and the command will be echoed again if the speed is 1200 baud or greater and it contains any control characters or less than one second has elapsed since the prompt was printed. The ESC character terminates canonical processing for the remainder of the command and the user can then modify the command line. This scheme has the advantages of canonical processing with the type-ahead echoing of raw mode. If the option **viraw** is also set, the terminal will always have canonical processing disabled. This mode is implicit for systems that do not support two alternate end-of-line delimiters, and may be helpful for certain terminals.

Input Edit Commands

erase	(User-defined erase character as defined by the *stty* command, usually ^**H** or #.) Delete previous character.
^**W**	Delete the previous blank separated word.
^**D**	Terminate the shell.
^**V**	Escape next character. Editing characters, the user's erase or kill characters may be entered in a command line or in a search string if preceded by a ^**V**. The ^**V** removes the next character's editing features (if any).
\	Escape the next *erase* or *kill* character.

Motion Edit Commands

These commands will move the cursor.

[*count*]**l**	Cursor forward (right) one character.
[*count*]**w**	Cursor forward one alpha-numeric word.
[*count*]**W**	Cursor to the beginning of the next word that follows a blank.
[*count*]**e**	Cursor to end of word.
[*count*]**E**	Cursor to end of the current blank delimited-word.
[*count*]**h**	Cursor backward (left) one character.
[*count*]**b**	Cursor backward one word.
[*count*]**B**	Cursor to preceding blank-separated word.
[*count*]**f***c*	Find the next character *c* in the current line.
[*count*]**F***c*	Find the previous character *c* in the current line.
[*count*]**t***c*	Equivalent to **f** followed by **h**.

[*count*]**T***c* Equivalent to **F** followed by **l**.

; Repeats the last single character find command, **f**, **F**, **t**, or **T**.

, Reverses the last single character find command.

0 Cursor to start of line.

^ Cursor to first non-blank character in line.

$ Cursor to end of line.

Search Edit Commands

These commands access your command history.

[*count*]**k** Fetch previous command. Each time **k** is entered the previous command back in time is accessed.

[*count*]– Equivalent to **k**.

[*count*]**j** Fetch next command. Each time **j** is entered the next command forward in time is accessed.

[*count*]+ Equivalent to **j**.

[*count*]**G** The command number *count* is fetched. The default is the least recent history command.

/string Search backward through history for a previous command containing *string*. *String* is terminated by a "RETURN" or "NEW LINE". If *string* is null the previous string will be used.

?*string* Same as / except that search will be in the forward direction.

n Search for next match of the last pattern to / or ? commands.

N Search for next match of the last pattern to / or ?, but in reverse direction.

/ Search history for the *string* entered by the previous / command.

Text Modification Edit Commands

These commands will modify the line.

a Enter input mode and enter text after the current character.

A Append text to the end of the line. Equivalent to $**a**.

[*count*]**c***motion*
c[*count*]*motion*
Delete current character through the character *motion* moves the cursor to and enter input mode. If *motion* is **c**, the entire line will be deleted and input mode entered.

C Delete the current character through the end of line and enter input mode. Equivalent to **c$**.

D Delete the current character through the end of line.

[*count*]**d***motion*
d[*count*]*motion*
Delete current character through the character *motion* moves the cursor to. Equivalent to **d$**. If *motion* is **d**, the entire line will be deleted.

i Enter input mode and insert text before the current character.

I Insert text before the beginning of the line. Equivalent to the two character sequence **^i**.

[*count*]**P** Place the previous text modification before the cursor.

[*count*]**p** Place the previous text modification after the cursor.

R Enter input mode and replace characters on the screen with characters you type overlay fashion.

r*c* Replace the current character with *c*.

[*count*]**x** Delete current character.

[*count*]**X** Delete preceding character.

[*count*]**.** Repeat the previous text modification command.

~ Invert the case of the current character and advance the cursor.

[*count*]**_** Causes the *count* word of the previous command to be appended and input mode entered. The last word is used if *count* is omitted.

***** Causes an ***** to be appended to the current word and file name generation attempted. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.

Other Edit Commands

Miscellaneous commands.

u Undo the last text modifying command.

U Undo all the text modifying commands performed on the line.

[*count*]**v** Returns the command **fc −e ${VISUAL:−${EDITOR:−vi}}** *count* in the input buffer. If *count* is omitted, then the current line is used.

^L	Line feed and print current line. Has effect only in control mode.
^J	(New line) Execute the current line, regardless of mode.
^M	(Return) Execute the current line, regardless of mode.
#	Equivalent to **I#**<*cr*>. Useful for causing the current line to be inserted in the history without being executed.

Special Commands.

The following commands are executed in the shell process. Input/Output redirection is permitted. File descriptor 1 is the default output location. Parameter assignment lists preceding the command do not remain in effect when the command completes unless noted.

: [*arg* **...**]
Parameter assignments remain in effect after the command completes. The command only expands parameters. A zero exit code is returned.

. *file* [*arg* **...**]
Parameter assignments remain in effect after the command completes. Read and execute commands from *file* and return. The commands are executed in the current Shell environment. The search path specified by **PATH** is used to find the directory containing *file*. If any arguments *arg* are given, they become the positional parameters. Otherwise the positional parameters are unchanged.

alias [**−tx**] [*name*[**=***value*] **...**]
Alias with no arguments prints the list of aliases in the form *name*=*value* on standard output. An *alias* is defined for each name whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution. The **−t** flag is used to set and list tracked aliases. The value of a tracked alias is the full pathname corresponding to the given *name*. The value becomes undefined when the value of **PATH** is reset but the aliases remain tracked. Without the **−t** flag, for each *name* in the argument list for which no *value* is given, the name and value of the alias is printed. The **−x** flag is used to set or print exported aliases. An exported alias is defined across sub-shell environments. **Alias** returns true unless a *name* is given for which no alias has been defined.

bg [*%job*]
This command is only built-in on systems that support job control. Puts the specified *job* into the background. The current job is put in the background if *job* is not specified.

break [*n*]
Exit from the enclosing **for**, **while**, **until**, or **select** loop,

if any. If *n* is specified then break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for**, **while**, **until**, or **select** loop. If *n* is specified then resume at the *n*th enclosing loop.

cd [*arg*]
cd *old new*
This command can be in either of two forms. In the first form it changes the current directory to *arg*. If *arg* is – the directory is changed to the previous directory. The shell parameter **HOME** is the default *arg*. The parameter **PWD** is set to the current directory. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is <**null**> (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / then the search path is not used. Otherwise, each directory in the path is searched for **arg**.

The second form of **cd** substitutes the string *new* for the string *old* in the current directory name, **PWD**, and tries to change to this new directory.

eval [*arg* **...**]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* **...**]
Parameter assignments remain in effect after the command completes. If *arg* is given, the command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and affect the current process. If no arguments are given the effect of this command is to modify file descriptors as prescribed by the input/output redirection list. In this case, any file descriptor numbers greater than 2 that are opened with this mechanism are closed when invoking another program.

exit [*n*]
Causes the 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 cause the shell to exit except for a shell which has the *ignoreeof* option (see **set** below) turned on.

export [*name* **...**]
The given *names* are marked for automatic export to the *environment* of subsequently-executed commands.

fc [**–e** *ename*] [**–nlr**] [*first*] [*last*]

fc **-e** - [*old*=*new*] [*command*]

In the first form, a range of commands from *first* to *last* is selected from the last **HISTSIZE** commands that were typed at the terminal. The arguments *first* and *last* may be specified as numbers or as strings. A string is used to locate the most recent command starting with the given string. A negative number is used as an offset to the current command number. If the flag **-l** is selected, the commands are listed on standard output. Otherwise, the editor program *ename* is invoked on a file containing these keyboard commands. If *ename* is not supplied, then the value of the parameter **FCEDIT** (default **/bin/ed**) is used as the editor. When editing is complete, the edited command(s) are executed. If *last* is not specified then it will be set to *first*. If *first* is not specified the default is the previous command for editing and -16 for listing. The flag **-r** reverses the order of the commands and the flag **-n** suppresses command numbers when listing. In the second form the *command* is re-executed after the substitution *old*=*new* is performed.

fg [*%job*]

This command is only built-in on systems that support job control. If *job* is specified it brings it to the foreground. Otherwise, the current job is brought into the foreground.

jobs [**-l**]

Lists the active jobs; given the **-l** option lists the process IDs in addition to the normal information.

kill [*-sig*] *process* **...**

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in **/usr/include/signal.h**, stripped of the prefix "SIG"). The signal names are listed by "kill -l". There is no default, saying just 'kill' does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal if it is stopped. The argument *process* can be either a process ID or a job.

let *arg* **...**

Each *arg* is an *arithmetic expression* to be evaluated. All calculations are done as long integers and no check for overflow is performed. Expressions consist of constants, named parameters, and operators. The following set of operators, listed in order of decreasing precedence, has been implemented:

-	unary minus
!	logical negation
* / %	multiplication, division, remainder

+ -	addition, subtraction
<= >= < >	comparison
==	equality
!=	inequality
=	arithmetic replacement

Sub-expressions in parentheses () are evaluated first and can be used to override the above precedence rules. The evaluation within a precedence group is from right to left for the = operator and from left to right for the others.

A parameter name must be a valid *identifier*. When parameter is encountered, the value associated with the parameter name is substituted and expression evaluation resumes. Up to nine levels of recursion are permitted.

The return code is 0 if the value of the last expression is non-zero, and 1 otherwise.

newgrp [*arg* **...**]
Equivalent to **exec newgrp** *arg* **...** .

print [**-Rnprsu**[*n*]] [*arg* **...**]
The Korn shell output mechanism. With no flags or with flag -, the arguments are printed on standard output as described by *echo*(1). In raw mode, **-R** or **-r**, the escape conventions of *echo* are ignored. The **-R** option will print all subsequent arguments and options other than **-n**. The **-p** option causes the arguments to be written onto the pipe of the process spawned with |& instead of standard output. The **-s** option causes the arguments to be written onto the history file instead of standard output. The **-u** flag can be used to specify a one digit file descriptor unit number **n** on which the output will be placed. The default is 1. If the flag **-n** is used, no **new-line** is added to the output.

read [**-prsu**[*n*]] [*name*?*prompt*] [*name* **...**]
The shell input mechanism. One line is read and is broken up into words using the characters in **IFS** as separators. In raw mode, **-r**, a \ at the end of a line does not signify line continuation. The first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. The **-p** option causes the input line to be taken from the input pipe of a process spawned by the shell using |&. If the **-s** flag is present, the input will be saved as a command in the history file. The flag **-u** can be used to specify a one-digit file descriptor unit to read from. The file descriptor can be opened with the **exec** special command. The default value of *n* is 0. If *name* is omitted then **REPLY** is used as the default *name*. The return code is 0 unless an end-of-file is encountered. An end-of-file with the **-p** option causes cleanup for this process so that another can be spawned. If the first argument contains a

?, the remainder of this word is used as a *prompt* when the shell is interactive. If the given file descriptor is open for writing and is a terminal device then the prompt is placed on this unit. Otherwise the prompt is issued on file descriptor 2. The return code is 0 unless an end-of-file is encountered.

readonly [*name* ...]
The given *names* are marked readonly and these names cannot be changed by subsequent assignment.

return [*n*]
Causes a shell *function* to return to the invoking script with the return status specified by *n*. If *n* is omitted then the return status is that of the last command executed. If **return** is invoked while not in a *function* then it is the same as an **exit**.

set [**−aefhkmnotuvx**] [**−o** *option* ...] [*arg* ...]
The flags for this command have meaning as follows:

−a All subsequent parameters that are defined are automatically exported.

−e If the shell is non-interactive and if a command fails, execute the **ERR** trap, if set, and exit immediately. This mode is disabled while reading profiles.

−f Disables file name generation.

−h Each command whose name is an *identifier* becomes a tracked alias when first encountered.

−k All parameter assignment arguments are placed in the environment for a command, not just those that precede the command name.

−m Background jobs will run in a separate process group and a line will print upon completion. The exit status of background jobs is reported in a completion message. On systems with job control, this flag is turned on automatically for interactive shells.

−n Read commands but do not execute them.

−o The following argument can be one of the following option names:

allexport Same as **−a**.

errexit Same as **−e**.

emacs Puts you in an *emacs* style in-line editor for command entry.

gmacs Puts you in a *gmacs* style in-line editor for command entry.

ignoreeof The shell will not exit on end-of-file. The command **exit** must be used.

keyword Same as **-k**.

markdirs All directory names resulting from file name genertion have a trailing / appended.

monitor Same as **-m**.

noexec Same as **-n**.

noglob Same as -f.

nounset Same as **-u**.

verbose Same as **-v**.

trackall Same as **-h**.

vi Puts you in insert mode of a *vi* style in-line editor until you hit escape (character **033**). This puts you in move mode. A return sends the line.

viraw Each character is processed as it is typed in *vi* mode.

xtrace Same as **-x**.

If no option name is supplied then the current option settings are printed.

-s Sort the positional parameters.

-t Exit after reading and executing one command.

-u Treat unset parameters as an error when substituting.

-v Print shell input lines as they are read.

-x Print comands and their arguments as they are executed.

- Turns off **-x** and **-v** flags and stops examining arguments for flags.

-- Do not change any of the flags; useful in setting **$1** to a value beginning with -. If no arguments follow this flag then the positional parameters are unset.

Using + rather than - causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in $-. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, **...**. If no arguments are given then the values of all names are printed on the standard output.

shift [*n*]

The positional parameters from $*n*+**1 ...** are renamed **$1 ...**, default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to $#.

test [*expr*]
Evaluate conditional expression *expr*. See *test*(1) for usage and description. The arithmetic comparison operators are not restricted to integers. They allow any arithmetic expression. Four additional primitive expressions are allowed:

-L *file* True if *file* is a symbolic link.

file1 **-nt** *file2*
True if *file1* is newer than *file2*.

file1 **-ot** *file2*
True if *file1* is older than *file2*.

file1 **-ef** *file2*
True if *file1* has the same device and i-node number as *file2*.

times Print the accumulated user and system times for the shell and for processes run from the shell.

trap [*arg*] [*sig*] **...**
Arg is a command to be read and executed when the shell receives signal(s) *sig*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Each *sig* can be given as a number or as the name of the signal. Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is omitted then all trap(s) *sig* are reset to their original values. If *arg* is the null string then this signal is ignored by the shell and by the commands it invokes. If *sig* is **ERR** then *arg* will be executed whenever a command has a non-zero exit code. This trap is not inherited by functions. If *sig* is 0 or **EXIT** and the **trap** statement is executed inside the body of a function, then the command *arg* is executed after the function completes. If *sig* is 0 or **EXIT** for a **trap** set outside any function then the command *arg* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

typeset [**-FLRZefilprtux**[*n*] [*name*[=*value*]] **...**]
Parameter assignments remain in effect after the command completes. When invoked inside a function, a new instance of the parameter *name* is created. The parameter value and type are restored when the function completes. The following list of attributes may be specified:

-F This flag provides UNIX to host-name file mapping on non-UNIX machines.

-L Left justify and remove leading blanks from *value*. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of first assignment. When the

parameter is assigned to, it is filled on the right with blanks or truncated, if necessary, to fit into the field. Leading zeros are removed if the **-Z** flag is also set. The **-R** flag is turned off.

-R Right justify and fill with leading blanks. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of the first assignment. The field is left filled with blanks or truncated from the end if the parameter is reassigned. The **-L** flag is turned off.

-Z Right justify and fill with leading zeros if the first non-blank character is a digit and the **-L** flag has not been set. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of the first assignment.

-e Tag the parameter as having an error. This tag is currently unused by the shell and can be set or cleared by the user.

-f The names refer to function names rather than parameter names. No assignments can be made and the only other valid flag is **-x**.

-i Parameter is an integer. This makes arithmetic faster. If *n* is non-zero it defines the output arithmetic base, otherwise the first assignment determines the output base.

-l All upper-case characters converted to lower-case. the upper-case flag, **-u**, is turned off.

-p The output of this command, if any, is written onto the two-way pipe.

-r The given *names* are marked read-only and these names cannot be changed by subsequent assignment.

-t Tags the named parameters. Tags are user definable and have no special meaning to the shell.

-u All lower-case characters are converted to upper-case characters. The lower-case flag, **-l**, is turned off.

-x The given *names* are marked for automatic export to the *environment* of subsequently executed commands.

Using + rather than - causes these flags to be turned off. If no *name* arguments are given but flags are specified, a list of *names* (and optionally the *values*) of the *parameters* which have these flags set is printed. (Using + rather than - keeps the values to be printed.) If no *names* and flags are given, the *names* and *attributes* of all *parameters* are printed.

ulimit [**-cdfmpt**] [*n*]

-c Imposes a size limit of *n* blocks on the size of core dumps (BSD only).

-d Imposes a size limit of *n* blocks on the size of the data area (BSD only).

-f Imposes a size limit of *n* blocks on files written by child processes (files of any size may be read).

-m Imposes a soft limit of *n* blocks on the size of physical memory (BSD only).

-p Changes the pipe size to *n* (UNIX/RT only).

-t Imposes a time limit of *n* seconds to be used by each process (BSD only).

If no option is given, **-f** is assumed. If *n* is not given the current limit is printed.

umask [*nnn*]
The user file-creation mask is set to *nnn* (see *umask*(2)). If *nnn* is omitted, the current value of the mask is printed.

unalias *name* **...**
The parameters given by the list of *names* are removed from the *alias* list.

unset [**-f**] *name* **...**
The parameters given by the list of *names* are unassigned, i.e., their values and attributes are erased. Read-only variables cannot be unset. If the flag **-f** is set, then the names refer to *function* names.

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.

whence [**-v**] *name* **...**
For each *name*, indicate how it would be interpreted if used as a command name. The flag **-v** produces a more verbose report.

Invocation.

If the Korn shell is invoked by *exec*(2), and the first character of argument zero (**$0**) is -, then the shell is assumed to be a *login* shell and commands are read from **/etc/profile** and then from either **.profile** in the current directory of **$HOME/.profile**, if either file exists. Next, commands are read from the file named by performing parameter substitution on the value of the environment parameter **ENV** if the 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, except for the output of some of the *special commands* listed above, 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 then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.

−r If the **−r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

EXIT STATUS

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 the **exit** command above).

FILES

/etc/passwd
/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

cat(1), cd(1), echo(1), env(1), newgrp(1), test(1), umask(1), vi(1), dup(2), exec(2), fork(2), pipe(2), signal(2), umask(2), ulimit(2), wait(2), rand(3C), a.out(4), profile(4), environ(5). If a command which is a *tracked alias* is executed, and then a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the **−t** option of the **alias** command to correct this situation.

If you move the current directory or one above it, **pwd** may not give the correct response. Use the **cd** command with a full path name to correct this situation.

Some very old shell scripts contain a ^ as a synonym for the pipe character |.

NAME

ld – link editor for common object files

SYNOPSIS

ld [**–e** epsym] [**–f** fill] [**–lx**] [**–m**] [**–o** outfile] [**–r**] [**–s**] [**–t**] [**–u** sysname] [**–x**] [**–Z**] [**–L** dir] [**–M**] [**–N**] [**–n**] [**–z**] [**–F**] [**–V**] [**–VS** num] [**–G**] [**–w**] file-names

DESCRIPTION

The *ld* command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and *ld* combines them, producing an object module that can either be executed or used as input for a subsequent *ld* run. The output of *ld* is left in **a.out**. This file is executable if no errors occurred during the load. If any input file, *file-name*, is not an object file, *ld* assumes it is either an ASCII file containing link editor directives or an archive library.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table (see *ar*(4)) is searched sequentially with as many passes as are necessary to resolve external references which can be satisfied by library members. Thus, the ordering of library members is unimportant.

The following options are recognized by *ld*.

–e epsym
: Set the default entry point address for the output file to be that of the symbol *epsym*.

–f fill
: Set the default fill pattern for "holes" within an output section as well as initialized *bss* sections. The argument *fill* is a two-byte constant.

–l*x*
: Search a library named **lib***x***.a** where *x* is up to nine characters. A library is searched when its name is encountered, so the placement of a **–l** is significant. By default, libraries are located in **/lib** and **/usr/lib**. However, if the shell variable **LIBROOT** is set, the value of **LIBROOT** is prepended to **/lib** and **/usr/lib** before searching the libraries.

–m
: Produce a map or listing of the input/output sections on the standard output.

–o outfile
: Produce an output object file by the name *outfile*. The name of the default object file is **a.out**.

–r
: Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent *ld* run. The link editor will not complain about unresolved references, and the output file will not be executed.

-s Strip line number entries and symbol table information from the output object file.

-t Turn off the warning about multiply-defined symbols that are not the same size.

-u symname
Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-x Do not preserve local (non-.global) symbols in the output symbol table; only enter external and static symbols. This option saves some space in the output file.

-Z Do not bind anything to address zero. This option will allow runtime detection of null pointers.

-L dir Change the algorithm of searching for **lib***x***.a** to look in *dir* before looking in /lib.

-M Output a message for each multiply-defined external definition. However, if the objects being loaded include debugging information, extraneous output is produced (see the **-g** option in *cc*(**1**)).

-N Put the data section immediately following the text in the output file. The result is a plain executable file, indicated by magic number 0407 in the operating system header.

-n Put the data section at the next segment boundary following the text section. The result is a shared text file, indicated by magic number 0410 in the operating system header.

-z Like **-n** but permits demand paged execution. This type of file is indicated by magic number 0413 in the operating system header.

-F Like **-z** but takes less disk space and can page faster into memory. This type is also indicated by magic number 0413 in the operating system header. It is distinguished by having virtual text and data starting addresses that are equal to the file offsets of the text and data sections, modulo 4096. The **-F** option is on by default.

-V Output a message giving information about the version of *ld* being used.

-VS num
Use **num** as a decimal version number identifying the **a.out** file that is produced. The version stamp is stored in the optional header.

-G Change the symbol name look-up algorithm as follows: if two names do not initially match, then if one of them is exactly eight characters, then a match is attempted only on the first eight characters. The purpose of this is to

allow compatibility between object modules that have been created with the old C compiler and with the new C compiler, which allows variable names more than eight characters long. A warning message is issued in such cases.

-w If **-G** is used, do not print warnings about symbols that partially matched.

FILES

/lib/lib*x*.a	libraries
/usr/lib/lib*x*.a	libraries
a.out	output file
/lib/ifile.0407	default **-N** directive file
/lib/ifile.0410	default **-n** directive file
/lib/ifile.0413	default **-z** directive file
/lib/ifile.0413-F	default **-F** directive file

SEE ALSO

as(1), cc(1), a.out(4), ar(4), exit(2), end(3C)

CAVEATS

Through its options and input directives, the common link editor gives users great flexibility; however, people who use the input directives must assume some added responsibilities. Input directives should insure the following properties for programs:

- C defines a zero pointer as null. A pointer to which zero has been assigned must not point to any object. To satisfy this, users must not place any object at virtual address zero in the data space.

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [**−rctvn**] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx−z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators ***, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, *?*, *+*, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character $ at the end of an expression requires a trailing new-line. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus **[a−zA−Z]+** matches a string of letters.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless(***p***)** pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext*+*yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a %%, as in YACC. Lines preceding %% which begin with a non-blank

character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

EXAMPLE

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
                switch (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                        default: go to loop;
                        }
                }
```

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

The flags must appear before any files. The flag **−r** indicates RATFOR actions, **−c** indicates C actions and is the default, **−t** causes the **lex.yy.c** program to be written instead to standard output, **−v** provides a one-line summary of statistics of the machine generated, **−n** will not print out the − summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default 2000)

%n *n* number of states is *n* (500)

%t *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions is *n* (3000)

The use of one or more of the above automatically implies the **−v** option, unless the **−n** option is used.

SEE ALSO

yacc(1).

LEX−Lexical Analyzer Generator by M. E. Lesk and E. Schmidt.

BUGS

The **−r** option is not yet fully operational.

NAME

line – read one line

SYNOPSIS

line

DESCRIPTION

Line copies one line (up to a new-line) from the standard input and writes it on the standard output. It returns an exit code of 1 on **EOF** and always prints at least a new-line. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1), read(2).

NAME

lint – a C program checker

SYNOPSIS

lint [**–abhlnpuvx**] file ...

DESCRIPTION

Lint attempts to detect features of the C program *files* which are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things which are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

It is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. By default, *lint* uses function definitions from the standard lint library **llib-lc.ln**; function definitions from the portable lint library **llib-port.ln** are used when *lint* is invoked with the **–p** option.

Any number of *lint* options may be used, in any order. The following options are used to suppress certain kinds of complaints:

–a Suppress complaints about assignments of long values to variables that are not long.

–b Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in a large number of such complaints.)

–h Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

–u Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program.)

–v Suppress complaints about unused arguments in functions.

–x Do not report variables referred to by external declarations but never used.

The following arguments alter *lint's* behavior:

–l*x* Include additional lint library **llib-l***x***.ln**. You can include a lint version of the math library **llib-lm.ln** by inserting **–lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. This option can be used to keep local lint libraries and is useful in the development of multi-file projects.

–n Do not check compatibility against either the standard or the portable lint library.

–p Attempt to check portability to other dialects (IBM and GCOS) of C.

The **-D**, **-U**, and **-I** options of *cc*(1) are also recognized as separate arguments.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
: at appropriate points stops comments about unreachable code.

/*VARARGS*n*/
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
: turns on the **-v** option for the next function.

/*LINTLIBRARY*/
: at the beginning of a file shuts off complaints about unused functions in this file.

Lint produces its first output on a per source file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

FILES

/usr/lib/lint[12]	programs
/usr/lib/llib-lc.ln	declarations for standard functions (binary format; source is in **/usr/lib/llib-lc**)
/usr/lib/llib-port.ln	declarations for portable functions (binary format; source is in **/usr/lib/llib-port**)
/usr/lib/llib-lm.ln	declarations for standard math functions (binary format; source is in **/usr/lib/llib-lm**)
/usr/tmp/*lint*	temporaries

SEE ALSO

cc(1).

BUGS

Exit(2) and other functions which do not return are not understood; this causes various lies.

NAME

logname – get login name

SYNOPSIS

logname

DESCRIPTION

Logname returns the contents of the environment variable **$LOGNAME**, which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1M), logname(3X), environ(5).

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder file ...

DESCRIPTION

The input is one or more object or library archive *files* (see *ar*(1)). The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1). Note that the link editor *ld*(1) is capable of multiple passes over an archive in the portable archive format (see *ar*(4)) and does not require that *lorder*(1) be used when building an archive. The usage of the *lorder*(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing **.o** files.

```
ar cr library `lorder *.o | tsort`
```

FILES

*symref, *symdef temporary files

SEE ALSO

ar(1), ld(1), tsort(1), ar(4).

BUGS

Object files whose names do not end with **.o**, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

NAME

lp, cancel – send/cancel requests to an LP line printer

SYNOPSIS

lp [**–c**] [**–d** dest] [**–m**] [**–n** number] [**–o** option] [**–r**] [**–t** title] [**–w**] [**–x**] files

cancel [**ids**] [**printers**] [**–a** name]

DESCRIPTION

Lp arranges for the named files and associated information (collectively called a *request*) to be printed by a line printer. If no file names are mentioned, the standard input is assumed. The file name – stands for the standard input and may be supplied on the command line in conjunction with named *files*. The order in which *files* appear is the same order in which they will be printed.

Lp associates a unique *id* with each request and prints it on the standard output. This *id* can be used later to cancel (see *cancel*) or find the status (see *lpstat*(1)) of the request.

The following options to *lp* may appear in any order and may be intermixed with file names:

–c Make copies of the *files* to be printed immediately when *lp* is invoked. Normally, *files* will not be copied, but will be linked whenever possible. If the **–c** option is not given, then the user should be careful not to remove any of the *files* before the request has been printed in its entirety. It should also be noted that in the absence of the **–c** option, any changes made to the named *files* after the request is made but before it is printed will be reflected in the printed output.

–d*dest* Choose *dest* as the printer or class of printers that is to do the printing. If *dest* is a printer, then the request will be printed only on that specific printer. If *dest* is followed by **_R**, the request is printed in raw mode (no post-processing, no CR-LF translation, high-order bits are passed through unchanged). If *dest* is a class of printers, then the request will be printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, etc.), requests for specific destinations may not be accepted (see *accept*(1M) and *lpstat*(1)). By default, *dest* is taken from the environment variable **LPDEST** (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems (see *lpstat*(1)).

–m Send mail (see *mail*(1)) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

–n*number* Print *number* copies (default of 1) of the output.

−o*option* Specify printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the **−o** keyletter more than once. For more information about what is valid for *options*, see *Models* in *lpadmin*(1M).

−r Remove file after printing.

−t*title* Print *title* on the banner page of the output.

−w Write a message on the user's terminal after the *files* have been printed. If the user is not logged in, then mail will be sent instead.

−x Display error message if print job is not sucessful. Display *id* if print job is successful. Note that *lp*(1) is silent by default.

−a*name* Assign *name* to print job. *Name* will display in name field by lpqueue.

Cancel cancels line printer requests that were made by the *lp*(1) command. The command line arguments may be either request *id*s (as returned by *lp*(1)) or printer *name*s (for a complete list, use *lpstat*(1)). Specifying a request *id* cancels the associated request even if it is currently printing. Specifying a printer *name* cancels the request which is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lpstat(1), mail(1).
accept(1M), lpadmin(1M), lpsched(1M) in the *UNIX System Administrator's Manual*.

BUGS

The −r, −w, and −a options are not available for Version 3.5

NAME

lpstat – print LP status information

SYNOPSIS

lpstat [options]

DESCRIPTION

Lpstat prints information about the current status of the LP line printer system.

If no *options* are given, then *lpstat* prints the status of all requests made to *lp*(1) by the user. Any arguments that are not *options* are assumed to be request *ids* (as returned by *lp*). *Lpstat* prints the status of such requests. *Options* may appear in any order and may be repeated and intermixed with other arguments. Some of the keyletters below may be followed by an optional *list* that can be in one of two forms: a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

```
-u"user1, user2, user3"
```

The omission of a *list* following such keyletters causes all information relevant to the keyletter to be printed, for example:

```
lpstat -o
```

prints the status of all output requests.

-a[*list*] Print acceptance status (with respect to *lp*) of destinations for requests. *List* is a list of intermixed printer names and class names.

-c[*list*] Print class names and their members. *List* is a list of class names.

-d Print the system default destination for *lp*.

-o[*list*] Print the status of output requests. *List* is a list of intermixed printer names, class names, and request *ids*.

-p[*list*] Print the status of printers. *List* is a list of printer names.

-r Print the status of the LP request scheduler

-s Print a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, and a list of printers and their associated devices.

-t Print all status information.

-u[*list*] Print status of output requests for users. *List* is a list of login names.

-v[*list*] Print the names of printers and the pathnames of the devices associated with them. *List* is a list of printer names.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1).

NAME

ls – list contents of directory

SYNOPSIS

ls [**–abcdfgilmqrstux1CFR**] name ...

DESCRIPTION

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

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

There are an unbelievable number of options:

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

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

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

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

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

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

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

–c Use time of last change of file status for sorting or printing.

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

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

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

-m Force stream output format

-1 Force one entry per line output format, e.g. to a teletype

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

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

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

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

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

-R Recursively list subdirectories encountered.

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

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

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

r if the file is readable;
w if the file is writable;
x if the file is executable;
s if the file is a special (super-user only) file;
- if the indicated permission is not granted.

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

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

The indications of set-ID and 1000 bit of the mode are capitalized (S and T respectively) if the corresponding execute permission is *not* set.

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

FILES

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

SEE ALSO

chmod(1).

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

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

Column widths choices are poor for terminals which can tab.

NAME

m4 – macro processor

SYNOPSIS

m4 [options] [files]

DESCRIPTION

M4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is –, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

–e Operate interactively. Interrupts are ignored and the output is unbuffered. Using this mode requires a special state of mind.

–s Enable line sync output for the C preprocessor (#line . . .)

–B*int* Change the size of the push-back and argument collection buffers from the default of 4,096.

–H*int* Change the size of the symbol table hash array from the default of 199. The size should be prime.

–S*int* Change the size of the call stack from the default of 100 slots. Macros take three slots, and non-macro arguments take one.

–T*int* Change the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any **–D** or **–U** flags:

–D*name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

–U*name*
undefines *name*.

Macro calls have the form:

name(arg1,arg2, . . ., argn)

The **(** must immediately follow the name of the macro. If the name of a defined macro is not followed by a **(**, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during

the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define — the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $n in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; $# is replaced by the number of arguments; $* is replaced by a list of all the arguments separated by commas; $@ is like $*, but each argument is quoted (with the current quotes).

undefine — removes the definition of the macro named in its argument.

defn — returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef — like *define*, but saves any previous definition.

popdef — removes current definition of its argument(s), exposing the previous one if any.

ifdef — if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word *unix* is predefined on UNIX versions of *m4*.

shift — returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote — change quote symbols to the first and second arguments. The symbols may be up to five characters long. *Changequote* without arguments restores the original values (i.e., ` ').

changecom — change left and right comment markers from the default # and new-line. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes new-line. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert — *m4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current

output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert — causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum — returns the value of the current output stream.

dnl — reads and discards characters up to and including the next new-line.

ifelse — has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr — returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr — returns the value of its argument decremented by 1.

eval — evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, −, *, /, %, ^ (exponentiation), bitwise &, |, ^, and ~; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len — returns the number of characters in its argument.

index — returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.

substr — returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit — transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include — returns the contents of the file named in the argument.

sinclude — is identical to *include*, except that it says nothing if the file is inaccessible.

syscmd — executes the UNIX command given in the first argument. No value is returned.

sysval	is the return code from the last call to *syscmd*.
maketemp	fills in a string of XXXXX in its argument with the current process ID.
m4exit	causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.
m4wrap	argument 1 will be pushed back at final EOF; example: m4wrap(`cleanup( )')
errprint	prints its argument on the diagnostic output file.
dumpdef	prints current names and definitions, for the named items, or for all if no arguments are given.
traceon	with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.
traceoff	turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

SEE ALSO

cc(1), cpp(1). *The M4 Macro Processor* by B. W. Kernighan and D. M. Ritchie.

NAME

mail, rmail – send mail to users or read mail

SYNOPSIS

mail [**-epqr**] [**-f** file]

mail [**-t**] persons

rmail [**-t**] persons

DESCRIPTION

Mail without arguments prints a user's mail, message-by-message, in last-in, first-out order. For each message, the user is prompted with a **?**, and a line is read from the standard input to determine the disposition of the message:

<new-line>	Go on to next message.
+	Same as <new-line>.
d	Delete message and go on to next message.
p	Print message again.
–	Go back to previous message.
s [*files*]	Save message in the named *files* (**mbox** is default).
w [*files*]	Save message, without its header, in the named *files* (**mbox** is default).
m [*persons*]	Mail the message to the named *persons* (yourself is default).
q	Put undeleted mail back in the *mailfile* and stop.
EOT (control-d)	Same as **q**.
x	Put all mail back in the *mailfile* unchanged and stop.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

The optional arguments alter the printing of the mail:

-e causes mail not to be printed. An exit value of 0 is returned if the user has mail; otherwise, an exit value of 1 is returned.

-p causes all mail to be printed without prompting for disposition.

-q causes *mail* to terminate after interrupts. Normally an interrupt only causes the termination of the message being printed.

-r causes messages to be printed in first-in, first-out order.

-f*file* causes *mail* to use *file* (e.g., **mbox**) instead of the default *mailfile*.

When *persons* are named, *mail* takes the standard input up to an end-of-file (or up to a line consisting of just a **.**) and adds it to

each *person*'s *mailfile*. The message is preceded by the sender's name and a postmark. Lines that look like postmarks in the message, (i.e., "From . . .") are preceded with a >. The **-t** option causes the message to be preceded by all *persons* the *mail* is sent to. A *person* is usually a user name recognized by *login*(1M). If a *person* being sent mail is not recognized, or if *mail* is interrupted during input, the file **dead.letter** will be saved to allow editing and resending.

To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1C)). Everything after the first exclamation mark in *persons* is interpreted by the remote system. In particular, if *persons* contains additional exclamation marks, it can denote a sequence of machines through which the message is to be sent on the way to its ultimate destination. For example, specifying **a!b!cde** as a recipient's name causes the message to be sent to user **b!cde** on system **a**. System **a** will interpret that destination as a request to send the message to user **cde** on system **b**. This might be useful, for instance, if the sending system can access system **a** but not system **b**, and system **a** has access to system **b**.

The *mailfile* may be manipulated in two ways to alter the function of *mail*. The *other* permissions of the file may be read-write, read-only, or neither read nor write to allow different levels of privacy. If changed to other than the default, the file will be preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

Forward to *person*

which will cause all mail sent to the owner of the *mailfile* to be forwarded to *person*. This is especially useful to forward all of a person's mail to one machine in a multiple machine environment.

Rmail only permits the sending of mail; *uucp*(1C) uses *rmail* as a security precaution.

When a user logs in, the presence of mail, if any, is indicated. Also, notification is made if new mail arrives while using *mail*.

FILES

/etc/passwd	to identify sender and locate persons
/usr/mail/*user*	incoming mail for *user*; i.e., the *mailfile*
$HOME/mbox	saved mail
$MAIL	variable containing path name of *mailfile*
/tmp/ma*	temporary file
/usr/mail/*.lock	lock for mail directory
dead.letter	unmailable text

SEE ALSO

login(1M), uucp(1C), write(1),
UNIX PC Electronic Mail User's Guide.

BUGS

Race conditions sometimes result in a failure to remove a lock file. After an interrupt, the next message may not be printed; printing may be forced by typing a **p**.

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [**–f** makefile] [**–p**] [**–i**] [**–k**] [**–s**] [**–r**] [**–n**] [**–b**] [**–e**] [**–m**] [**–t**] [**–d**] [**–q**] [names]

DESCRIPTION

The following is a brief description of all options and some special names:

–f *makefile* Description file name. *Makefile* is assumed to be the name of a description file. A file name of – denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

–p Print out the complete set of macro definitions and target descriptions.

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

–k Abandon work on the current entry, but continue on other branches that do not depend on that entry.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ are printed.

–b Compatibility mode for old makefiles.

–e Environment variables override assignments within makefiles.

–m Print a memory map showing text, data, and stack. This option is a no-operation on systems without the *getu* system call.

–t Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

–d Debug mode. Print out detailed information on files and times examined.

–q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the **-s** option.

.IGNORE Same effect as the **-i** option.

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **-f** option is present, **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** are tried in order. If *makefile* is -, the standard input is taken. More than one **-** *makefile* argument pair may appear.

Make updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out of date.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a **:**, then a (possibly null) list of prerequisite files or dependencies. Text following a **;** and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><new-line> sequence. Everything printed by make (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a\
b
```

will produce

```
ab
```

exactly the same as the shell would.

Sharp (#) and new-line surround comments.

The following *makefile* says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h**:

```
pgm: a.o b.o
        cc a.o b.o -o pgm
a.o: incl.h a.c
        cc -c a.c
b.o: incl.h b.c
        cc -c b.c
```

Command lines are executed one at a time, each by its own shell. The first one or two characters in a command can be the following: -, **@**, **-@**, or **@-**. If **@** is present, printing of the command is suppressed. If **-** is present, *make* ignores an error. A line is printed when it is executed unless the **-s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the initial character sequence contains a **@**. The **-n** option specifies printing without execution; however, if the command line has the string **$(MAKE)** in it, the line is always executed (see discussion of the

MAKEFLAGS macro under *Environment*). The **-t** (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate *make*. If the **-i** option is present, or the entry **.IGNORE:** appears in *makefile*, or the initial character sequence of the command contains -. the error is ignored. If the **-k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **-b** option allows old makefiles (those written for the old version of *make*) to run without errors. The difference between the old version of *make* and this version is that this version requires all dependency lines to have a (possibly null or implicit) command associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

Interrupt and quit cause the target to be deleted unless the target is a dependency of the special name **.PRECIOUS**.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The **-e** option causes the environment to override the macro assignments in a makefile.

The **MAKEFLAGS** environment variable is processed by *make* as containing any legal input option (except **-f**, **-p**, and **-d**) defined for the command line. Further, upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the **-n** option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make -n** recursively on a whole software system to see what would have been executed. This is because the **-n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped newline. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional :*subst1*=*subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*.

Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, new-line characters, and beginnings of lines. An example of the use of the substitute sequence is shown under *Libraries*.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

$* The macro $* stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@ The $@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$< The $< macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out of date with respect to the target (i.e., the "manufactured" dependent file name). Thus, in the **.c.o** rule, the $< macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc -c -O $*.c
```

or:

```
.c.o:
        cc -c -O $<
```

$? The $? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules which must be rebuilt.

$% The $% macro is only evaluated when the target is an archive library member of the form **lib(file.o)**. In this case, $@ evaluates to **lib** and $% evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, **./** is generated. The only macro excluded from this alternative form is $?. The reasons for this are debatable.

Suffixes

Certain names (for instance, those ending with **.o**) have inferable prerequisites such as **.c**, **.s**, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h

The internal rules for *make* are contained in the source file **rules.c** for the *make* program. These rules can be locally modified. To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

make -fp - 2>/dev/null </dev/null

The only peculiarity in this output is the **(null)** string which *printf*(3S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file (see *sccsfile*(4)). Thus, the rule **.c~.o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix it is incompatible with *make*'s suffix point-of-view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e. **.c:**) is the definition of how to build *x* from *x***.c**. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

.SUFFIXES: .o .c .y .l .s

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o -o pgm
a.o b.o: incl.h
```

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*(1), *lex*(1), and *yacc*(1) respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with **.c.o:** as the target and no dependents. Shell commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parenthesis, it is assumed to be an archive library, the string within parenthesis referring to a member within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library which contains **file.o**. (This assumes the **LIB** macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form ***.XX.a*** where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(file.o)** depend upon **file.o** explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

In fact, the **.c.a** rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?  @echo lib is now up to date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object file names (inside **lib**) whose C source files are out of date. The substitution mode translates the **.o** to **.c**. (Unfortunately, one cannot as yet transform to **.c~**; however, this may become possible in the future.) Note also, the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

EXAMPLE

The following makefile segment demonstrates a feature that allows makefiles to include other makefile segments:

```
#Makefile to build world
include $(MAKEINC) Makepre.h

foo:  fooa.o    foob.o
      $(cc)     fooa.o  foob.o  -o  foo


include $(MAKEINC)/Makepost.h
```

In the above example, the environment variable MAKEINC will be expanded and used as the directory where the file Makepre.h and Makepost.h exist.

FILES

[Mm]akefile and s.[Mm]akefile

SEE ALSO

sh(1).
Make–A Program for Maintaining Computer Programs by S. I. Feldman.
An Augmented Version of Make by E. G. Bradford.

BUGS

Some commands return non-zero status inappropriately; use **–i** to overcome the difficulty. Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across new-lines in *make*. The syntax **(lib(file1.o file2.o file3.o)** is illegal. You cannot build **lib(file.o)** from **file.o**. The macro **$(a:.o=.c~)** doesn't work.

NAME

makekey – generate encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

This feature is available only in the domestic (U.S.) version of the UNIX PC software. *Makekey* improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e., to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, ., /, and upper- and lower-case letters. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4,096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but broken in 4,096 different ways. Using the *input key* as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 *output key* bits in the result.

Makekey is intended for programs that perform encryption (e.g., *ed*(1) and *crypt*(1)). Usually, its input and output will be pipes.

SEE ALSO

crypt(1), ed(1), passwd(4).

NAME

mesg – permit or deny messages

SYNOPSIS

mesg [**n**] [**y**]

DESCRIPTION

Mesg with argument **n** forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reports the current state without changing it.

FILES

/dev/tty*

SEE ALSO

write(1).

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

message – display error and help messages

SYNOPSIS

message [**-u**] [**-c**] [**-i**] text

DESCRIPTION

Message allows the shell programmer access to the *message*(3T) subroutine. *Text* is a text string with the standard special character conventions: **\n** for newline, etc.

The possible options are:

-u Use the current window for the messages—resizes it to fit.

-c Create a confirmation message (see MT_CONFIRM in *message*(3T)).

-i Create a *pop-up* message—press any key to return to the caller (see MT_POPUP in *message*(3T)).

If no options are set, *message*(1) will generate an error message (see MT_ERROR in *message*(3T)).

EXAMPLES

The following example prints a confirmation message using the current window:

```
message -uc "Do you wish to continue"
        if [ "$?" != "0" ]
        then
                exit
        fi
```

SEE ALSO

message(3T), shform(1), tam(3T).

NAME

mkdir – make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777 (possibly altered by *umask*(1)). Standard entries, **.**, for the directory itself, and **..**, for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

sh(1), rm(1), umask(1).

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

NAME

mm, osdd, checkmm – print/check documents formatted with the MM macros

SYNOPSIS

mm [options] [files]

osdd [options] [files]

checkmm [files]

DESCRIPTION

Mm can be used to type out documents using *nroff* and the MM text-formatting macro package. It has options to specify preprocessing by *tbl*(1) and/or *neqn* (see *eqn*(1)) and postprocessing by various terminal-oriented output filters. The proper pipelines and the required arguments and flags for *nroff* and MM are generated, depending on the options selected.

Osdd is equivalent to the command **mm –mosd**.

Options for *mm* are given below. Any other arguments or flags (e.g., **–rC3**) are passed to *nroff* or to MM, as appropriate. Such options can occur in any order, but they must appear before the *files* arguments. If no arguments are given, *mm* prints a list of its options.

–T*term* Specifies the type of output terminal; for a list of recognized values for *term*, type **help term2**. If this option is *not* used, *mm* will use the value of the shell variable **$TERM** from the environment (see *profile*(4) and *environ*(5)) as the value of *term*, if **$TERM** is set; otherwise, *mm* will use **450** as the value of *term*. If several terminal types are specified, the last one takes precedence.

–12 Indicates that the document is to be produced in 12-pitch. May be used when **$TERM** is set to one of **300**, **300s**, **450**, and **1620**. (The pitch switch on the DASI 300 and 300s terminals must be manually set to **12** if this option is used.)

–c Causes *mm* to invoke *col*(1); note that *col*(1) is invoked automatically by *mm* unless *term* is one of **300**, **300s**, **450**, **37**, **4000a**, **382**, **4014**, **tek**, **1620**, and **X**.

–e Causes *mm* to invoke *neqn*; also causes *neqn* to read the **/usr/pub/eqnchar** file (see *eqnchar*(5)).

–t Causes *mm* to invoke *tbl*(1).

–E Invokes the **–e** option of *nroff*.

–y Causes *mm* to use the non-compacted version of the macros (see *mm*(5)).

As an example (assuming that the shell variable **$TERM** is set in the environment to **450**), the two command lines below are equivalent:

```
mm –t –rC3 –12 ghh*
tbl ghh* | nroff –cm –T450–12 –h –rC3
```

Mm reads the standard input when – is specified instead of any file names. (Mentioning other files together with – leads to disaster.) This option allows *mm* to be used as a filter, e.g.:

```
cat dws | mm -
```

Checkmm is a program for checking the contents of the named *files* for errors in the use of the Memorandum Macros, missing or unbalanced *neqn* delimiters, and **.EQ/.EN** pairs. Note: The user need not use the *checkeq* program (see *eqn*(1)). Appropriate messages are produced. The program skips all directories, and if no file name is given, standard input is read.

HINTS

1. *Mm* invokes *nroff* with the **–h** flag. With this flag, *nroff* assumes that the terminal has tabs set every 8 character positions.
2. Use the **–o***list* option of *nroff* to specify ranges of pages to be output. Note, however, that *mm*, if invoked with one or more of the **–e**, **–t**, and – options, *together* with the **–o***list* option of *nroff* may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.
3. If you use the **–s** option of *nroff* (to stop between pages of output), use line-feed (rather than return or new-line) to restart the output. The **–s** option of *nroff* does not work with the **–c** option of *mm*, or if *mm* automatically invokes *col*(1) (see **–c** option above).
4. If you lie to *mm* about the kind of terminal its output will be printed on, you'll get (often subtle) garbage; however, if you are redirecting output into a file, use the **–T37** option, and then use the appropriate terminal filter when you actually print that file.

SEE ALSO

col(1), cw(1), env(1), eqn(1), greek(1), nroff(1), tbl(1), profile(4), mm(5), term(5).
UNIX System Document Processing Guide.

DIAGNOSTICS

mm "mm: no input file" if none of the arguments is a readable file and *mm* is not used as a filter.

checkmm "Cannot open *filename*" if file(s) is unreadable. The remaining output of the program is diagnostic of the source file.

NAME

mmt, mvt – typeset documents, view graphs, and slides

SYNOPSIS

mmt [options] [files]

mvt [options] [files]

DESCRIPTION

These two commands are very similar to *mm*(1), except that they both typeset their input via *troff* (not included on the UNIX PC), as opposed to formatting it via *nroff*; *mmt* uses the MM macro package, while *mvt* uses the Macro Package for View Graphs and Slides. These two commands have options to specify preprocessing by *tbl*(1) and/or *eqn*(1). The proper pipelines and the required arguments and flags for *troff* and for the macro packages are generated, depending on the options selected.

Options are given below. Any other arguments or flags (e.g., **–rC3**) are passed to *troff* or to the macro package, as appropriate. Such options can occur in any order, but they must appear before the *files* arguments. If no arguments are given, these commands print a list of their options.

–e	Causes these commands to invoke *eqn*(1); also causes *eqn* to read the **/usr/pub/eqnchar** file (see *eqnchar*(5)).
–t	Causes these commands to invoke *tbl*(1).
–Tst	Directs the output to the MH STARE facility.
–Tvp	Directs the output to a Versatec printer; this option is not available at all UNIX sites.
–T4014	Directs the output to a Tektronix 4014 terminal via the *tc*(1) filter.
–Ttek	Same as **–T4014**.
–a	Invokes the **–a** option of *troff*.
–y	Causes *mmt* to use the non-compacted version of the macros (see *mm*(5)). No effect for *mvt*.

These commands read the standard input when – is specified instead of any file names.

Mvt is just a link to *mmt*.

HINT

Use the **–o***list* option of *troff* to specify ranges of pages to be output. Note, however, that these commands, if invoked with one or more of the **–e**, **–t**, and – options, *together* with the **–o***list* option of *troff* may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

SEE ALSO

env(1), eqn(1), mm(1), tbl(1), tc(1), profile(4), environ(5), mm(5).
UNIX System Document Processing Guide.

DIAGNOSTICS

"m[mv]t: no input file" if none of the arguments is a readable file and the command is not used as a filter.

NAME

more, page – file perusal filter for crt viewing

SYNOPSIS

more [**-cdflsu**] [*-n*] [*+linenumber*] [*+/pattern*] [name ...]

page *more options*

DESCRIPTION

More is a filter which allows examination of a continuous text one screen full (or window full) at a time on a soft-copy terminal. It normally pauses after each screen full, printing **--More--** at the bottom of the screen. If the user then types a carriage return, one more line is displayed. If the user hits a space, another screen full is displayed. Other possibilities are enumerated later.

The command line options are:

-n An integer which is the size (in lines) of the window which *more* will use instead of the default.

-c *More* will draw each page by beginning at the top of the screen and erasing each line just before it draws on it. This avoids scrolling the screen, making it easier to read while *more* is writing. This option will be ignored if the terminal does not have the ability to clear to the end of a line.

-d *More* will prompt the user with the message *Hit space to continue, Rubout to abort* at the end of each screen full. This is useful if *more* is being used as a filter in some setting, such as a class, where many users may be unsophisticated.

-f This causes *more* to count logical lines, rather than screen lines. That is, long lines are not folded. This option is recommended if *nroff* output is being piped through *ul*, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen positions, but which do not print when they are sent to the terminal as part of an escape sequence. Thus *more* may think that lines are longer than they actually are, and fold lines erroneously.

-l Do not treat ^L (form feed) specially. If this option is not given, *more* will pause after any line that contains a ^L, as if the end of a screen full had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

-s Squeeze multiple blank lines from the output, producing only one blank line. Especially helpful when viewing *nroff* output, this option maximizes the useful information present on the screen.

-u Normally, *more* will handle underlining such as produced by *nroff* in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a

stand-out mode, *more* will output appropriate escape sequences to enable underlining or stand-out mode for underlined information in the source file. The *-u* option suppresses this processing.

+linenumber
Start up at *linenumber*.

+/pattern
Start up two lines before the line containing the regular expression *pattern*.

If the program is invoked as *page*, then the screen is cleared before each screen full is printed (but only if a full screen is being printed), and *k* - 1 rather than *k* - 2 lines are printed in each screen full, where *k* is the number of lines the terminal can display.

More looks in the *TERMCAP* environment variable or the file **/etc/termcap** to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

More looks in the environment variable *MORE* to pre-set any flags desired. For example, if you prefer to view files using the *-c* mode of operation, the *csh* command *setenv MORE -c* or the *sh* command sequence *MORE='-c' ; export MORE* would cause all invocations of *more*, including invocations by programs such as *man* and *msgs*, to use this mode. Normally, the user will place the command sequence which sets up the *MORE* environment variable in the **.cshrc** or **.profile** file.

If *more* is reading from a file, rather than a pipe, then a percentage is displayed along with the **--More--** prompt. This gives the fraction of the file (in characters, not lines) that has been read so far.

Other sequences which may be typed when *more* pauses, and their effects, are as follows (*i* is an optional integer argument, defaulting to 1):

i<space>
display *i* more lines, (or another screen full if no argument is given)

^D display 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i*.

d same as ^D (control-D)

*i*z same as typing a space except that *i*, if present, becomes the new window size.

*i*s skip *i* lines and print a screen full of lines

*i*f skip *i* screen fulls and print a screen full of lines

q or Q Exit from *more*.

= Display the current line number.

v Start up the editor *vi* at the current line.

h Help command; give a description of all the *more* commands.

i/expr search for the *i*th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screen full is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

*i*n search for the *i*th occurrence of the last regular expression entered.

' (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!command
invoke a shell with *command*. The characters % and ! in *command* are replaced with the current file name and the previous shell command respectively. If there is no current file name, % is not expanded. The sequences \% and \! are replaced by % and ! respectively.

i:n skip to the *i*th next file given in the command line (skips to last file if *n* doesn't make sense).

i:p skip to the *i*th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

:f display the current file name and line number.

:q or :Q
exit from *more* (same as q or Q).

. (dot) repeat the previous command.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command character itself is given, the user may hit the line kill character to cancel the numerical argument being formed. In addition, the user may hit the erase character to redisplay the **--More--(xx%)** message.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\). *More* will stop sending output, and will display the usual **--More--** prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to *noecho* mode by this program so that the output can be continuous. What you type will thus not show on your terminal, except for the / and ! commands.

If the standard output is not a teletype, then *more* acts just like *cat*, except that a header is printed before each file (if there is more than one).

A sample usage of *more* in previewing *nroff* output would be

```
nroff -ms +2 doc.n | more -s
```

FILES

/etc/termcap	Terminal data base
/usr/lib/more.help	Help file

SEE ALSO

sh(1), environ(5).

NAME

newform – change the format of a text file

SYNOPSIS

newform [**–s**] [**–i** tabspec] [**–o** tabspec] [**–b** n] [**–e** n] [**–p** n] [**–a** n] [**–f**] [**–c** char] [**–l** n] [files]

DESCRIPTION

Newform reads lines from the named *files*, or the standard input if no input file is named, and reproduces the lines on the standard output. Lines are reformatted in accordance with command line options in effect.

Except for **–s**, command line options may appear in any order, may be repeated, and may be intermingled with the optional *files*. Command line options are processed in the order specified. This means that option sequences like "**–e**15 **–l**60" will yield results different from "**–l**60 **–e**15". Options are applied to all *files* on the command line.

–i*tabspec* Input tab specification: expands tabs to spaces, according to the tab specifications given. *Tabspec* recognizes all tab specification forms described in *tabs*(1). In addition, *tabspec* may be **––**, in which *newform* assumes that the tab specification is to be found in the first line read from the standard input (see *fspec*(4)). If no *tabspec* is given, *tabspec* defaults to **–8**. A *tabspec* of **–0** expects no tabs; if any are found, they are treated as **–1**.

–o*tabspec* Output tab specification: replaces spaces by tabs, according to the tab specifications given. The tab specifications are the same as for **–i***tabspec*. If no *tabspec* is given, *tabspec* defaults to **–8**. A *tabspec* of **–0** means that no spaces will be converted to tabs on output.

–l*n* Set the effective line length to *n* characters. If *n* is not entered, **–l** defaults to 72. The default line length without the **–l** option is 80 characters. Note that tabs and backspaces are considered to be one character (use **–i** to expand tabs to spaces).

–b*n* Truncate *n* characters from the beginning of the line when the line length is greater than the effective line length (see **–l***n*). Default is to truncate the number of characters necessary to obtain the effective line length. The default value is used when **–b** with no *n* is used. This option can be used to delete the sequence numbers from a COBOL program as follows:

newform –l1 –b7 file-name

The **–l1** must be used to set the effective line length shorter than any existing line in the file so that the **–b** option is activated.

–e*n* Same as **–b***n* except that characters are truncated from the end of the line.

−c*k* Change the prefix/append character to *k*. Default character for *k* is a space.

−p*n* Prefix *n* characters (see **−c***k*) to the beginning of a line when the line length is less than the effective line length. Default is to prefix the number of characters necessary to obtain the effective line length.

−a*n* Same as **−p***n* except characters are appended to the end of a line.

−f Write the tab specification format line on the standard output before any other lines are output. The tab specification format line which is printed will correspond to the format specified in the *last* **−o** option. If no **−o** option is specified, the line which is printed will contain the default specification of **−8**.

−s Shears off leading characters on each line up to the first tab and places up to 8 of the sheared characters at the end of the line. If more than 8 characters (not counting the first tab) are sheared, the eighth character is replaced by a * and any characters to the right of it are discarded. The first tab is always discarded.

An error message and program exit will occur if this option is used on a file without a tab on each line. The characters sheared off are saved internally until all other options specified are applied to that line. The characters are then added at the end of the processed line.

For example, to convert a file with leading digits, one or more tabs, and text on each line, to a file beginning with the text, all tabs after the first expanded to spaces, padded with spaces out to column 72 (or truncated to column 72), and the leading digits placed starting at column 73, the command would be:

newform −s −i −l −a −e file-name

DIAGNOSTICS

All diagnostics are fatal.

usage: . . .	*Newform* was called with a bad option.
not −s format	There was no tab on one line.
can't open file	Self-explanatory.
internal line too long	A line exceeds 512 characters after being expanded in the internal work buffer.
tabspec in error	A tab specification is incorrectly formatted, or specified tab stops are not ascending.
tabspec indirection illegal	A *tabspec* read from a file (or standard input) may not contain a *tabspec* referencing another file (or standard input).

EXIT CODES

0 – normal execution
1 – for any error

SEE ALSO

csplit(1), tabs(1), fspec(4).

BUGS

Newform normally only keeps track of physical characters; however, for the **–i** and **–o** options, *newform* will keep track of backspaces in order to line up tabs in the appropriate logical columns.

Newform will not prompt the user if a *tabspec* is to be read from the standard input (by use of **–i**– – or **–o**– –).

If the **–f** option is used, and the last **–o** option specified was **–o**– –, and was preceded by either a **–o**– – or a **–i**– –, the tab specification format line will be incorrect.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp [–] [group]

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login*(1M). The same person remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

Newgrp without an argument changes the group identification to the group in the password file; in effect it changes the group identification back to the caller's original group.

An initial – flag causes the environment to be changed to the one that would be expected if the user actually logged in again.

A password is demanded if the group has a password and the user himself does not, or if the group has a password and the user is not listed in **/etc/group** as being a member of that group.

When most users log in, they are members of the group named **other**.

FILES

/etc/group
/etc/passwd

SEE ALSO

login(1M), group(4).

BUGS

There is no convenient way to enter a password into **/etc/group**. Use of group passwords is not encouraged, because, by their very nature, they encourage poor security practices. Group passwords may disappear in the future.

NAME

nice – run a command at low priority

SYNOPSIS

nice [–increment] command [arguments]

DESCRIPTION

Nice executes *command* with a lower CPU scheduling priority. If the *increment* argument (in the range 1-19) is given, it is used; if not, an increment of 10 is assumed.

The super-user may run commands with priority higher than normal by using a negative increment, e.g., **––10**.

SEE ALSO

nohup(1), nice(2).

DIAGNOSTICS

Nice returns the exit status of the subject command.

BUGS

An *increment* larger than 19 is equivalent to 19.

NAME

nl – line numbering filter

SYNOPSIS

nl [–**h**type] [–**b**type] [–**f**type] [–**v**start#] [–**i**incr] [–**p**] [–**l**num] [–**s**sep] [–**w**width] [–**n**format] [–**d**delim] file

DESCRIPTION

Nl reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

Nl views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (e.g. no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line contents	*Start of*
\:\:\:	header
\:\:	body
\:	footer

Unless optioned otherwise, *nl* assumes the text being read is in a single logical page body.

Command options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

–**b***type* Specifies which logical page body lines are to be numbered. Recognized *types* and their meaning are: **a**, number all lines; **t**, number lines with printable text only; **n**, no line numbering; **p***string*, number only lines that contain the regular expression specified in *string*. Default *type* for logical page body is **t** (text lines numbered).

–**h***type* Same as –**b***type* except for header. Default *type* for logical page header is **n** (no lines numbered).

–**f***type* Same as –**b***type* except for footer. Default for logical page footer is **n** (no lines numbered).

–**p** Do not restart numbering at logical page delimiters.

–**v***start#* *Start#* is the initial value used to number logical page lines. Default is **1**.

–**i***incr* *Incr* is the increment value used to number logical page lines. Default is **1**.

-s*sep* *Sep* is the character(s) used in separating the line number and the corresponding text line. Default *sep* is a tab.

-w*width* *Width* is the number of characters to be used for the line number. Default *width* is **6**.

-n*format* *Format* is the line numbering format. Recognized values are: **ln**, left justified, leading zeroes suppressed; **rn**, right justified, leading zeroes suppressed; **rz**, right justified, leading zeroes kept. Default *format* is **rn** (right justified).

-l*num* *Num* is the number of blank lines to be considered as one. For example, **-l2** results in only the second adjacent blank being numbered (if the appropriate **-ha**, **-ba**, and/or **-fa** option is set). Default is **1**.

-d*xx* The delimiter characters specifying the start of a logical page section may be changed from the default characters (\:) to two user specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the **-d** and the delimiter characters. To enter a backslash, use two backslashes.

EXAMPLE

The command:

```
nl -v10 -i10 -d!+ file1 file2
```

will number files 1 and 2 starting at line number 10 with an increment of ten. The logical page delimiters are !+.

SEE ALSO

pr(1).

NAME

nm – print name list of common object file

SYNOPSIS

nm [**–o**] [**–x**] [**–h**] [**–v**] [**–n**] [**–e**] [**–f**] [**–u**] [**–V**] [**–T**] file-names

DESCRIPTION

The *nm* command displays the symbol table of each common object file *file-name*. *File-name* may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, the following information will be printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Class Its storage class.

Type Its type and derived type. If the symbol is an instance of a structure or of a union then the structure or union tag will be given following the type (e.g. struct-tag). If the symbol is an array, then the array dimensions will be given following the type (eg., **char**[n][m]). Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Size Its size in bytes, if available. Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Line The source line number at which it is defined, if available. Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Section For storage classes static and external, the object file section containing the symbol (e.g., text, data or bss).

The output of *nm* may be controlled using the following options:

–o Print the value and size of a symbol in octal instead of decimal.

–x Print the value and size of a symbol in hexadecimal instead of decimal.

–h Do not display the output header data.

–v Sort external symbols by value before they are printed.

–n Sort external symbols by name before they are printed.

–e Print only external and static symbols.

–f Produce full output. Print redundant symbols (.text, .data and .bss), normally suppressed.

–u Print undefined symbols only.

–V Print the version of the *nm* command executing on the standard error output.

–T By default, *nm* prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The **–T** option causes *nm* to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **nm name –e –v** and **nm –ve name** print the static and external symbols sorted by value.

FILES

/usr/tmp/nm??????

CAVEATS

When all the symbols are printed, they must be printed in the order they appear in the symbol table in order to preserve scoping information. Therefore, the **–v** and **–n** options should be used only in conjunction with the **–e** option.

SEE ALSO

as(1), cc(1), ld(1), a.out(4), ar(4).

DIAGNOSTICS

"nm: name: cannot open"
if *name* cannot be read.

"nm: name: bad magic"
if *name* is not an appropriate common object file.

"nm: name: no symbols"
if the symbols have been stripped from *name*.

NAME

nohup – run a command immune to hangups and quits

SYNOPSIS

nohup command [arguments]

DESCRIPTION

Nohup executes *command* with hangups and quits ignored. If output is not re-directed by the user, it will be sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to **$HOME/nohup.out**.

SEE ALSO

nice(1), signal(2).

NAME

nroff – format text

SYNOPSIS

nroff [options] [files]

DESCRIPTION

Nroff formats text contained in *files* (standard input by default) for printing on typewriter-like devices and line printers. Its capabilities are described in the *NROFF/ TROFF User's Manual* cited below.

An argument consisting of a minus (–) is taken to be a file name corresponding to the standard input. The *options*, which may appear in any order, but must appear before the *files*, are:

–o*list* Print only pages whose page numbers appear in the *list* of numbers and ranges, separated by commas. A range *N–M* means pages *N* through *M*; an initial *–N* means from the beginning to page *N*; and a final *N–* means from *N* to the end. (See *BUGS* below.)

–n*N* Number first generated page *N*.

–s*N* Stop every *N* pages. *Nroff* will halt *after* every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a line-feed or new-line (new-lines do not work in pipelines, e.g., with *mm*(1)). This option does not work if the output of *nroff* is piped through *col*(1). When *nroff* halts between pages, an ASCII **BEL** is sent to the terminal.

–r*aN* Set register *a* (which must have a one-character name) to *N*.

–i Read standard input after *files* are exhausted.

–q Invoke the simultaneous input-output mode of the **.rd** request.

–z Print only messages generated by **.tm** (terminal message) requests.

–m*name* Prepend to the input *files* the non-compacted (ASCII text) macro file **/usr/lib/tmac/tmac.***name*.

–c*name* Prepend to the input *files* the compacted macro files **/usr/lib/macros/cmp.**[**nt**].[**dt**].*name* and **/usr/lib/macros/ucmp.**[**nt**].*name*.

–k*name* Compact the macros used in this invocation of *nroff*, placing the output in files [**dt**].*name* in the current directory (see the May 1979 Addendum to the *NROFF/ TROFF User's Manual* for details of compacting macro files).

–T*name* Prepare output for specified terminal. Known *name*s are **37** for the (default) TELETYPE Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **300s** for the DASI 300s, **300** for the DASI 300, **450** for the DASI 450, **lp** for a

(generic) ASCII line printer, **382** for the DTC-382, **4000A** for the Trendata 4000A, **832** for the Anderson Jacobson 832, **X** for a (generic) EBCDIC printer, and **2631** for the Hewlett Packard 2631 line printer.

-e Produce equally-spaced words in adjusted lines, using the full resolution of the particular terminal.

-h Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

-u*n* Set the emboldening factor (number of character overstrikes) for the third font position (bold) to *n*, or to zero if *n* is missing.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta$#	temporary file
/usr/lib/tmac/tmac.*	standard macro files and pointers
/usr/lib/macros/*	standard macro files
/usr/lib/term/*	terminal driving tables for *nroff*

SEE ALSO

NROFF/ TROFF User's Manual
A TROFF Tutorial
col(1), eqn(1), greek(1), mm(1), tbl(1), mm(5).

BUGS

Nroff believes in Eastern Standard Time; as a result, depending on the time of the year and on your local time zone, the date that *nroff* generates may be off by one day from your idea of what the date is.

When *nroff* is used with the *-olist* option inside a pipeline (e.g., with one or more of *eqn*(1) and *tbl*(1)), it may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

NAME

od – octal dump

SYNOPSIS

od [**–bcdosx**] [file] [[+]offset[**.**][**b**]]

DESCRIPTION

Od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing, **–o** is default. The meanings of the format options are:

–b Interpret bytes in octal.

–c Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=**\0**, backspace=**\b**, form-feed=**\f**, new-line=**\n**, return=**\r**, tab=**\t**; others appear as 3-digit octal numbers.

–d Interpret words in unsigned decimal.

–o Interpret words in octal.

–s Interpret 16-bit words in signed decimal.

–x Interpret words in hex.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended, the offset is interpreted in decimal. If **b** is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded by +.

Dumping continues until end-of-file.

SEE ALSO

dump(1).

NAME

pack, pcat, unpack – compress and expand files

SYNOPSIS

pack [–] name ...

pcat name ...

unpack name ...

DESCRIPTION

Pack attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file *name* is replaced by a packed file *name*.**z** with the same access modes, access and modified dates, and owner as those of *name*. If *pack* is successful, *name* will be removed. Packed files can be restored to their original form using *unpack* or *pcat*.

Pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the – argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of – in place of *name* will cause the internal flag to be set and reset.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each **.z** file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Load modules, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

Pack returns a value that is the number of files that it failed to compress.

No packing will occur if:

the file appears to be already packed;
the file name has more than 12 characters;
the file has links;
the file is a directory;
the file cannot be opened;
no disk storage blocks will be saved by packing;
a file called *name*.**z** already exists;
the **.z** file cannot be created;
an I/O error occurred during processing.

The last segment of the file name must contain no more than 12 characters to allow space for the appended **.z** extension. Directories cannot be compressed.

Pcat does for packed files what *cat*(1) does for ordinary files. The specified files are unpacked and written to the standard output. Thus to view a packed file named *name*.**z** use:

pcat name.z

or just:

pcat name

To make an unpacked copy, say *nnn*, of a packed file named *name*.**z** (without destroying *name*.**z**) use the command:

pcat name >nnn

Pcat returns the number of files it was unable to unpack. Failure may occur if:

the file name (exclusive of the **.z**) has more than 12 characters;
the file cannot be opened;
the file does not appear to be the output of *pack*.

Unpack expands files created by *pack*. For each file *name* specified in the command, a search is made for a file called *name*.**z** (or just *name*, if *name* ends in **.z**). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the **.z** suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

Unpack returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in *pcat*, as well as for the following:

a file with the "unpacked" name already exists;
if the unpacked file cannot be created.

NAME

passwd – change login password

SYNOPSIS

passwd name

DESCRIPTION

This command changes (or installs) a password associated with the login *name*.

The program prompts for the old password (if any) and then for the new one (twice). The caller must supply these. New passwords should be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. Only the first eight characters of the password are significant.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password. Only the super-user can create a null password.

The password file is not changed if the new password is the same as the old password, or if the password has not "aged" sufficiently; see *passwd*(4).

FILES

/etc/passwd

SEE ALSO

login(1M), crypt(3C), passwd(4).

NAME

paste – merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste file1 file2 . . .
paste **–d** list file1 file2 . . .
paste **–s** [**–d** list] file1 file2 . . .

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). If you will, it is the counterpart of *cat*(1) which concatenates vertically, i.e., one file after the other. In the last form above, *paste* subsumes the function of an older command with the same name by combining subsequent lines of the input file (serial merging). In all cases, lines are glued together with the *tab* character, or with characters from an optionally specified *list*. Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if – is used in place of a file name.

The meanings of the options are:

–d Without this option, the new-line characters of each but the last file (or last line in case of the **–s** option) are replaced by a *tab* character. This option allows replacing the *tab* character by one or more alternate characters (see below).

list One or more characters immediately following **–d** replace the default *tab* as the line concatenation character. The list is used circularly, i. e. when exhausted, it is reused. In parallel merging (i. e. no **–s** option), the lines from the last file are always terminated with a new-line character, not from the *list*. The list may contain the special escape sequences: **\n** (new-line), **\t** (tab), **** (backslash), and **\0** (empty string, not a null character). Quoting may be necessary, if characters have special meaning to the shell (e.g. to get one backslash, use *–d "\\\\"*).

–s Merge subsequent lines rather than one from each input file. Use *tab* for concatenation, unless a *list* is specified with **–d** option. Regardless of the *list*, the very last character of the file is forced to be a new-line.

– May be used in place of any file name, to read a line from the standard input. (There is no prompting).

EXAMPLES

ls | paste –d" " –	list directory in one column
ls | paste – – – –	list directory in four columns
paste –s –d"\ t\ n" file	combine pairs of lines into lines

SEE ALSO

grep(1), cut(1),

pr(1): **pr −t −m**... works similarly, but creates extra blanks, tabs and new-lines for a nice page layout.

DIAGNOSTICS

line too long	Output lines are restricted to 511 characters.
too many files	Except for **−s** option, no more than 12 input files may be specified.

NAME

path – locate executable file for command

SYNOPSIS

path *command*

DESCRIPTION

Path is a quick way to discover what executable file is behind a shell command. It searches each directory mentioned in your **PATH** environment variable until it finds an executable file called *command.*

NAME

pr – print files

SYNOPSIS

pr [options] [files]

DESCRIPTION

Pr prints the named files on the standard output. If *file* is –, or if no files are specified, the standard input is assumed. By default, the listing is separated into pages, each headed by the page number, a date and time, and the name of the file.

By default, columns are of equal width, separated by at least one space; lines which do not fit are truncated. If the **–s** option is used, lines are not truncated and columns are separated by the separation character.

If the standard output is associated with a terminal, error messages are withheld until *pr* has completed printing.

The below *options* may appear singly or be combined in any order:

+*k* Begin printing with page *k* (default is 1).

–*k* Produce *k*-column output (default is 1). The options **–e** and **–i** are assumed for multi-column output.

–a Print multi-column output across the page.

–m Merge and print all files simultaneously, one per column (overrides the –*k*, and **–a** options).

–d Double-space the output.

–e*ck* Expand *input* tabs to character positions *k*+1, 2**k*+1, 3**k*+1, etc. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If *c* (any non-digit character) is given, it is treated as the input tab character (default for *c* is the tab character).

–i*ck* In *output*, replace white space wherever possible by inserting tabs to character positions *k*+1, 2**k*+1, 3**k*+1, etc. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. If *c* (any non-digit character) is given, it is treated as the output tab character (default for *c* is the tab character).

–n*ck* Provide *k*-digit line numbering (default for *k* is 5). The number occupies the first *k*+1 character positions of each column of normal output or each line of **–m** output. If *c* (any non-digit character) is given, it is appended to the line number to separate it from whatever follows (default for *c* is a tab).

–w*k* Set the width of a line to *k* character positions (default is 72 for equal-width multi-column output, no limit otherwise).

-o*k* Offset each line by *k* character positions (default is 0). The number of character positions per line is the sum of the width and offset.

-l*k* Set the length of a page to *k* lines (default is 66).

-h Use the next argument as the header to be printed instead of the file name.

-p Pause before beginning each page if the output is directed to a terminal (*pr* will ring the bell at the terminal and wait for a carriage return).

-f Use form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.

-r Print no diagnostic reports on failure to open files.

-t Print neither the five-line identifying header nor the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.

-s*c* Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

EXAMPLES

Print **file1** and **file2** as a double-spaced, three-column listing headed by "file list":

```
pr -3dh "file list" file1 file2
```

Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, 37, . . . :

```
pr -e9 -t <file1 >file2
```

FILES

/dev/tty* to suspend messages

SEE ALSO

cat(1).

NAME

prof – display profile data

SYNOPSIS

prof [**–tcan**] [**–ox**] [**–g**] [**–z**] [**–h**] [**–s**] [**–m** mdata] [prog]

DESCRIPTION

Prof interprets the profile file produced by the *monitor*(3C) function. The symbol table in the object file *prog* (**a.out** by default) is read and correlated with the profile file (**mon.out** by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options **t, c, a,** and **n** determine the type of sorting of the output lines:

–t Sort by decreasing percentage of total time (default).

–c Sort by decreasing number of calls.

–a Sort by increasing symbol address.

–n Sort lexically by symbol name.

The mutually exclusive options **o** and **x** specify the printing of the address of each symbol monitored:

–o Print each symbol address (in octal) along with the symbol name.

–x Print each symbol address (in hexadecimal) along with the symbol name.

The following options may be used in any combination:

–g Include non-global symbols (static functions).

–z Include all symbols in the profile range (see *monitor*(3C)), even if associated with zero number of calls and zero time.

–h Suppress the heading normally printed on the report. (This is useful if the report is to be processed further.)

–s Print a summary of several of the monitoring parameters and statistics on the standard error output.

–m mdata
Use file *mdata* instead of **mon.out** for profiling data.

For the number of calls to a function to be tallied, the **–p** option of *cc*(1) must have been given when the file containing the function was compiled. This option to the *cc* command also arranges for the object file to include a special profiling start-up function that calls *monitor*(3C) at the beginning and end of execution. It is the call to *monitor* at the end of execution that causes the **mon.out** file to be written. Thus, only programs that call *exit*(2) or return from *main* will cause the **mon.out** file to be produced.

FILES

mon.out for profile
a.out for namelist

SEE ALSO

cc(1), nm(1), exit(2), profil(2), monitor(3C).

BUGS

There is a limit of 300 functions that may have call counters established during program execution. If this limit is exceeded, other data will be overwritten and the **mon.out** file will be corrupted. The number of call counters used will be reported automatically by the *prof* command whenever the number exceeds 250.

NAME

prs – print an SCCS file

SYNOPSIS

prs [**–d**[dataspec]] [**–r**[SID]] [**–e**] [**–l**] [**–a**] files

DESCRIPTION

Prs prints, on the standard output, parts or all of an SCCS file (see *sccsfile*(4)) in a user supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**), and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of *keyletter* arguments, and file names.

All the described *keyletter* arguments apply independently to each named file:

–d[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *DATA KEYWORDS*) interspersed with optional user supplied text.

–r[*SID*] Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

–e Requests information for all deltas created *earlier* than and including the delta designated via the **–r** keyletter.

–l Requests information for all deltas created *later* than and including the delta designated via the **–r** keyletter.

–a Requests printing of information for both removed, i.e., delta type = *R*, (see *rmdel*(1)) and existing, i.e., delta type = *D*, deltas. If the **–a** keyletter is not specified, information for existing deltas only is provided.

DATA KEYWORDS

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(4)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is

either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User supplied text is any text other than recognized data keywords. A tab is specified by **\t** and carriage return/new-line is specified by **\n**.

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D* or *R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS: ...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS: ...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS: ...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes* or *no*	S
:BF:	Branch flag	"	*yes* or *no*	S
:J:	Joint edit flag	"	*yes* or *no*	S
:LK:	Locked releases	"	:R: ...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

TABLE 1 (Continued)

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file

may produce on the standard output:

Users and/or user IDs for s.file are:
xyz
131
abc

prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:" -r s.file

may produce on the standard output:

Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas

As a *special case:*

prs s.file

may produce on the standard output:

D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta

for each delta table entry of the "D" type. The only keyletter argument allowed to be used with the *special case* is the **-a** keyletter.

FILES

/tmp/pr?????

SEE ALSO

admin(1), delta(1), get(1), help(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ps – report process status

SYNOPSIS

ps [options]

DESCRIPTION

Ps prints certain information about active processes. Without *options*, information is printed about processes associated with the current terminal. Otherwise, the information that is displayed is controlled by the following *options*:

–e Print information about all processes.

–d Print information about all processes, except process group leaders.

–a Print information about all processes, except process group leaders and processes not associated with a terminal.

–f Generate a *full* listing. (Normally, a short listing containing only process ID, terminal ("tty") identifier, cumulative execution time, and the command name is printed.) See below for meaning of columns in a full listing.

–l Generate a *long* listing. See below.

–c *corefile* Use the file *corefile* in place of **/dev/mem**.

–s *swapdev* Use the file *swapdev* in place of **/dev/swap**. This is useful when examining a *corefile*; a *swapdev* of **/dev/null** will cause the user block to be zeroed out.

–n *namelist* The argument will be taken as the name of an alternate *namelist* (**/unix** is the default).

–t *tlist* Restrict listing to data about the processes associated with the terminals given in *tlist*, where *tlist* can be in one of two forms: a list of terminal identifiers separated from one another by a comma, or a list of terminal identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

–p *plist* Restrict listing to data about processes whose process ID numbers are given in *plist*, where *plist* is in the same format as *tlist*.

–u *ulist* Restrict listing to data about processes whose user ID numbers or login names are given in *ulist*, where *ulist* is in the same format as *tlist*. In the listing, the numerical user ID will be printed unless the **–f** option is used, in which case the login name will be printed.

-g *glist* Restrict listing to data about processes whose process groups are given in *glist*, where *glist* is a list of process group leaders and is in the same format as *tlist*.

The column headings and the meaning of the columns in a *ps* listing are given below; the letters **f** and **l** indicate the option (*full* or *long*) that causes the corresponding heading to appear; **all** means that the heading always appears. Note that these two options only determine what information is provided for a process; they do *not* determine which processes will be listed.

F (l) Flags (octal and additive) 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;
40	another tracing flag.

S (l) The state of the process:

0	non-existent;
S	sleeping;
W	waiting;
R	running;
I	intermediate;
Z	terminated;
T	stopped;
X	growing.

UID (f,l) The user ID number of the process owner; the login name is printed under the **-f** option.

PID (all) The process ID of the process; it is possible to kill a process if you know this datum.

PPID (f,l) The process ID of the parent process.

C (f,l) Processor utilization for scheduling.

STIME (f) Starting time of the process.

PRI (l) The priority of the process; higher numbers mean lower priority.

NI (l) Nice value; used in priority computation.

ADDR (l) The memory address of the process (a pointer to the segment table array on the 3B20S), if resident; otherwise, the disk address.

SZ (l) The size in blocks of the core image of the process.

WCHAN	(l)	The event for which the process is waiting or sleeping; if blank, the process is running.
TTY	(all)	The controlling terminal for the process.
TIME	(all)	The cumulative execution time for the process.
CMD	(all)	The command name; the full command name and its arguments are printed under the **-f** option.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked <**defunct**>.

Under the **-f** option, *ps* tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the **-f** option, is printed in square brackets.

FILES

/unix	system namelist.
/dev/mem	memory.
/dev/swap	the default swap device.
/etc/passwd	supplies UID information.
/etc/ps_data	internal data structure.
/dev	searched to find terminal ("tty") names.

SEE ALSO

kill(1), nice(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 are irrelevant.

NAME

ptx – permuted index

SYNOPSIS

ptx [options] [input [output]]

DESCRIPTION

Ptx generates the file *output* that can be processed with a text formatter to produce a permuted index of file *input* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of each line. *Ptx* output is in the form:

.xx "tail" "before keyword" "keyword and after" "head"

where **.xx** is assumed to be an *nroff* or *troff* macro provided by the user, or provided by the *mptx*(5) macro package. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed. *Tail* and *head*, at least one of which is always the empty string, are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line.

The following *options* can be applied:

–f Fold upper and lower case letters for sorting.

–t Prepare the output for the phototypesetter.

–w *n* Use the next argument, *n*, as the length of the output line. The default line length is 72 characters for *nroff* and 100 for *troff*.

–g *n* Use the next argument, *n*, as the number of characters that *ptx* will reserve in its calculations for each gap among the four parts of the line as finally printed. The default gap is 3.

–o *only* Use as keywords only the words given in the *only* file.

–i *ignore* Do not use as keywords any words given in the *ignore* file. If the **–i** and **–o** options are missing, use **/usr/lib/eign** as the *ignore* file.

–b *break* Use the characters in the *break* file to separate words. Tab, new-line, and space characters are *always* used as break characters.

–r Take any leading non-blank characters of each input line to be a reference identifier (as to a page or chapter), separate from the text of the line. Attach that identifier as a 5th field on each output line.

The index for this manual was generated using *ptx*.

FILES

/bin/sort
/usr/lib/eign
/usr/lib/tmac/tmac.ptx

SEE ALSO

nroff(1), mm(5), mptx(5).

BUGS

Line length counts do not account for overstriking or proportional spacing.

Lines that contain tildes (~) are botched, because *ptx* uses that character internally.

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the path name of the working (current) directory.

SEE ALSO

cd(1).

DIAGNOSTICS

"Cannot open .." and "Read error in .." indicate possible file system trouble and should be referred to a UNIX programming counselor.

NAME

regcmp – regular expression compile

SYNOPSIS

regcmp [–] files

DESCRIPTION

Regcmp, in most cases, precludes the need for calling *regcmp*(3X) from C programs. This saves on both execution time and program size. The command *regcmp* compiles the regular expressions in *file* and places the output in *file*.**i**. If the – option is used, the output will be placed in *file*.**c**. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotes. The output of *regcmp* is C source code. Compiled regular expressions are represented as **extern char** vectors. *File*.**i** files may thus be *included* into C programs, or *file*.**c** files may be compiled and later loaded. In the C program which uses the *regcmp* output, *regex*(*abc*,*line*) will apply the regular expression named *abc* to *line*. Diagnostics are self-explanatory.

EXAMPLES

name "([A–Za–z][A–Za–z0–9_]*)$0"

telno "\({0,1}([2–9][01][1–9])$0\){0,1} *"
"([2–9][0–9]{2})$1[–]{0,1}"
"([0–9]{4})$2"

In the C program that uses the *regcmp* output,

regex(telno, line, area, exch, rest)

will apply the regular expression named *telno* to *line*.

SEE ALSO

regcmp(3X).

NAME

rm, rmdir – remove files or directories

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** option is given or if the standard input is not a terminal.

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

rmdel – remove a delta from an SCCS file

SYNOPSIS

rmdel –r*SID* files

DESCRIPTION

Rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the delta specified must *not* be that of a version being edited for the purpose of making a delta (i. e., if a *p-file* (see *get*(1)) exists for the named SCCS file, the delta specified must *not* appear in any entry of the *p-file*).

If a directory is named, *rmdel* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The exact permissions necessary to remove a delta are documented in the *Source Code Control System User's Guide*. Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

delta(1), get(1), help(1), prs(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sact – print current SCCS file editing activity

SYNOPSIS

sact files

DESCRIPTION

Sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(1) with the **–e** option has been previously executed without a subsequent execution of *delta*(1). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1	specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.
Field 2	specifies the SID for the new delta to be created.
Field 3	contains the logname of the user who will make the delta (i.e. executed a *get* for editing).
Field 4	contains the date that **get –e** was executed.
Field 5	contains the time that **get –e** was executed.

SEE ALSO

delta(1), get(1), unget(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff **–r**SID1 **–r**SID2 [**–p**] [**–s**n] files

DESCRIPTION

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

–r*SID?*	*SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to *bdiff*(1) in the order given.
–p	pipe output for each file through *pr*(1).
–s*n*	*n* is the file segment size that *bdiff* will pass to *diff*(1). This is useful when *diff* fails due to a high system load.

FILES

/tmp/get????? Temporary files

SEE ALSO

bdiff(1), get(1), help(1), pr(1).
Source Code Control System User's Guide
UNIX System User's Guide.

DIAGNOSTICS

"*file*: No differences" If the two versions are the same.
Use *help*(1) for explanations.

NAME

scrset – set screen save time

SYNOPSIS

scrset [*n*]

DESCRIPTION

Scrset enables and disables the screen save feature. When enabled, this feature causes the screen to go blank after a given interval of time has elapsed with no keyboard or mouse input; the next keystroke or mouse motion restores the screen display. This is a new feature of the UNIX PC 3.0 release.

The parameter *n*, if greater than 0, is the number of seconds to delay before turning off the screen. *N* equal to 0 turns off the screen save feature (this is the default condition). If *n* is less than 0, the screen is immediately turned off.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [–**w**] [–**W**] [objfil [corfil [directory-list]]]

DESCRIPTION

Sdb is a symbolic debugger that can be used with C programs. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

Objfil is normally an executable program file which has been compiled with the –**g** (debug) option; if it has not been compiled with the –**g** option, or if it is not an executable file, the symbolic capabilities of *sdb* will be limited, but the file can still be examined and the program debugged. The default for *objfil* is **a.out**. *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core**. The core file need not be present. A – in place of *corfil* will force *sdb* to ignore any core image file. The colon separated list of directories (*directories-list*) is used to locate the source files used to build *objfil*.

It is useful to know that at any time there is a *current line* and *current file*. If *corfil* exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in *main*(). The current line and file may be changed with the source file examination commands.

Initially *sdb* has a greater-than character (>) prompt, which indicates that *sdb* is ready for the user to enter the first command. After *sdb* has begun, the prompt is <*x*>, where *x* is the name of the last command given.

By default, warnings are provided if the source files used in producing *objfil* cannot be found, or are newer than *objfil*. This checking feature and the accompanying warnings may be disabled by the use of the –**W** flag.

Names of variables are written just as they are in C. Note that names in C are now of arbitrary length, *sdb* will no longer truncate names. Variables local to a procedure may be accessed using the form *procedure*:*variable*. If no procedure name is given, the procedure containing the current line is used by default.

It is also possible to refer to structure members as *variable*.*member*, pointers to structure members as *variable*–>*member* and array elements as *variable*[*number*]. Pointers may be dereferenced by using the form *pointer*[0]. Combinations of these forms may also be used. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure referenced by *sdb*. An unqualified structure variable may also be used with various commands. Generally, *sdb* will interpret a structure as a set of variables. Thus, *sdb* will display the values of all the elements of a structure when it is requested to display a structure.

An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value *sdb* displays, not the addresses of individual elements.

Elements of a multidimensional array may be referenced as

variable[*number*][*number*]...,

or as

variable[*number*,*number*, ...].

In place of *number*, the form *number*;*number* may be used to indicate a range of values, * may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. As with structures, *sdb* displays all the values of an array or section of an array if trailing subscripts are omitted. It displays only the address of the array itself or section specified by the user if subscripts are omitted.

A particular instance of a variable on the stack may be referenced by using the form *procedure*:*variable*,*number*. All the variations mentioned in naming variables may be used. *Number* is the occurrence of the specified procedure on the stack, counting the top, or most current, as the first. If no procedure is specified, the procedure currently executing is used by default.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as *filename*:*number* or *procedure*:*number*. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under *sdb*, all addresses refer to the executing program; otherwise they refer to *objfil* or *corfil*. An initial argument of **-w** permits overwriting locations in *objfil*.

Addresses.

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

$b1 \leq address < e1$

file address $= address + f1 - b1$

otherwise

$b2 \leq address < e2$

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.

The initial setting of both mappings is suitable for normal **a.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.

In order for *sdb* to be used on large files all appropriate values are kept as signed 32-bit integers.

Commands.

The commands for examining data in the program are:

t Print a stack trace of the terminated or halted program.

T Print the top line of the stack trace.

variable/*clm*

Print the value of *variable* according to length *l* and format *m*. A numeric count *c* indicates that a region of memory, beginning at the address implied by *variable*, is to be displayed. The length specifiers are:

b one byte
h two bytes (half word)
l four bytes (long word)

Legal values for *m* are:

c character
d decimal
u decimal, unsigned
o octal
x hexadecimal
f 32-bit single precision floating point
g 64-bit double precision floating point
s Assume *variable* is a string pointer and print characters starting at the address pointed to by the variable.
a Print characters starting at the variable's address. This format may not be used with register variables.
p pointer to procedure
i disassemble machine language instruction with addresses printed symbolically.
I disassemble machine language instruction with addresses just printed numerically.

The length specifiers are only effective with the formats[**c**, **d**, **u**, **o** and **x**. Any of the specifiers *c*, *l*, and *m* may be omitted. If all are omitted, *sdb* chooses a length and a format suitable for the variable's type as declared in the program. If *m* is specified, then this format is used for displaying the variable. A length specifier determines the output length of the value to be displayed, sometimes resulting in truncation. A count specifier *c* tells *sdb* to display that many units of memory, beginning at the address of *variable*. The number of bytes in one such unit of memory is determined

by the length specified *l*, or if no length is given, by the size associated with the *variable*. If a count specifier is used for the **s** or **a** command, then that many characters are printed. Otherwise successive characters are printed until either a null byte is reached or 128 characters are printed. The last variable may be redisplayed with the command **./**.

The *sh*(1) metacharacters * and ? may be used within procedure and variable names, providing a limited form of pattern matching. If no procedure name is given, both variables local to the current procedure and global variables are matched; if a procedure name is specified then only variables local to that procedure are matched. To match only global variables, the form *:pattern* is used.

linenumber?*lm*
variable:?*lm*

Print the value at the address from **a.out** or I space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is 'i'.

variable=*lm*
linenumber=*lm*
number=*lm*

Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

variable!*value*

Set *variable* to the given *value*. The value may be a number, character constant or a variable. The value must be well defined; expressions which produce more than one value, such as structures, are not allowed. Character constants are denoted *'character*. Numbers are viewed as integers unless a decimal point or exponent is used. In this case, they are treated as having the type double. Registers are viewed as integers. The *variable* may be an expression which indicates more than one variable, such as an array or structure name. If the address of a variable is given, it is regarded as the address of a variable of type *int*. C conventions are used in performing any type conversions necessary to perform the indicated assignment.

f Print the 68881 floating-point registers.

x Print the machine registers and the current machine-language instruction.

X Print the current machine-language instruction.

The commands for examining source files are:

e *procedure*
e *file-name*
e *directory/*

e *directory file-name*
The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure and file names are reported.

/regular expression/
Search forward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing / may be omitted.

?regular expression?
Search backward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing ? may be deleted.

p Print the current line.

z Print the current line followed by the next 9 lines. Set the current line to the last line printed.

w Window. Print the 10 lines around the current line.

number
Set the current line to the given line number. Print the new current line.

count+
Advance the current line by *count* lines. Print the new current line.

count–
Retreat the current line by *count* lines. Print the new current line.

The commands for controlling the execution of the source program are:

count **r** *args*
count **R**
Run the program with the given arguments. The **r** command with no arguments reuses the previous arguments to the program while the **R** command runs the program with no arguments. An argument beginning with < or > causes redirection for the standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber **c** *count*
linenumber **C** *count*
Continue after a breakpoint or interrupt. If *count* is given, it specifies the number of breakpoints to be ignored. **C** continues with the signal which caused the program to stop and **c** ignores it. If a *linenumber* is specified then a temporary breakpoint is placed at the line and execution is continued. This temporary breakpoint is deleted when the command

finishes.

linenumber **g** *count*
Continue after a breakpoint with execution resumed at the given line. If *count* is given, it specifies the number of breakpoints to be ignored.

s *count*
S *count*
Single step the program through *count* lines. If no count is given then the program is run for one line. **S** is equivalent to **s** except it steps through procedure calls.

i
I Single step by one machine language instruction. **I** steps with the signal which caused the program to stop reactivated and **i** ignores it.

variable$**m** *count*
address:**m** *count*
Single step (as with **s**) until the specified location is modified with a new value. If *count* is omitted, it is effectively infinity. *Variable* must be accessible from the current procedure. Since this command is done by software, it can be very slow.

level **v**
Toggle verbose mode, for use when single stepping with **S**, **s** or **m**. If *level* is omitted, then just the current source file and/or subroutine name is printed when either changes. If *level* is 1 or greater, each C source line is printed before it is executed; if *level* is 2 or greater, each assembler statement is also printed. A **v** turns verbose mode off if it is on for any level.

k Kill the program being debugged.

procedure(arg1,arg2,...)
procedure(arg1,arg2,...)/*m*
Execute the named procedure with the given arguments. Arguments can be integer, character or string constants or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to **d**.

linenumber **b** *commands*
Set a breakpoint at the given line. If a procedure name without a line number is given (e.g. "proc:"), a breakpoint is placed at the first line in the procedure even if it was not compiled with the **–g** option. If no *linenumber* is given, a breakpoint is placed at the current line. If no *commands* are given then execution stops just before the breakpoint and control is returned to *sdb*. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If **k** is used as a command to execute at a breakpoint, control returns to *sdb*, instead of continuing

execution.

B Print a list of the currently active breakpoints.

linenumber **d**
Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively: each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d** then the breakpoint is deleted.

D Delete all breakpoints.

l Print the last executed line.

linenumber **a**
Announce. If *linenumber* is of the form *proc*:*number*, the command effectively does a *linenumber* **b l**. If *linenumber* is of the form *proc*:, the command effectively does a *proc*: **b T**.

Miscellaneous commands:

!*command*
The command is interpreted by *sh*(1).

new-line
Perform the previous command again.

control-D
Scroll. Print the next 10 lines of instructions, source or data depending on which was printed last.

< *filename*
Read commands from *filename* until the end of file is reached, and then continue to accept commands from standard input. When *sdb* is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

M Print the address maps.

M [?/] [*] *b e f*
Record new values for the address map. The arguments **?** and **/** specify the text and data maps respectively. The first segment, (*b1*,*e1*,*f1*), is changed unless * is specified, in which case the second segment (*b2*,*e2*,*f2*) of the mapping is changed. If fewer than three values are given, the remaining map parameters are left unchanged.

" *string*
Print the given string. The C escape sequences of the form *character* are recognized, where *character* is a nonnumeric character.

q Exit the debugger.

The following commands also exist and are intended only for debugging the debugger:

V Print the version number.
Q Print a list of procedures and files being debugged.
Y Toggle debug output.

Sdb may be instructed to monitor a given memory location and stop the program when the value at that location changes in a given way. For example:

> if x < = 123

The above example instructs *sdb* to monitor the value at location *x*. When the user gives the command to continue (**c**), *sdb* checks the value of x at every source line executed and stops the program if the given condition becomes true. Note that use of this constraint slows the real-time execution of a program.

The syntax of the *if* command is as follows:

if Shows a list of the current data breakpoints; assigns a number to each.

if *var* Monitors the value of *var* and stops the program if the value changes. A variable name may be used for *var,* as well as a constant address. Comparisons are done as either 4-byte signed or 4-byte unsigned, depending on the data type. To perform a 1-byte or 2-byte comparison, an optional length value may accompany *var.* An example of a 2-byte comparison is

> if x,2 = 0xff

if *var rel value*
Compares the value of *var* to the constant given and stops the program if the condition is true. The values of *rel* may be =, ==, <, <=, >, >=, or !=.

off *n* Disables or turns off a data breakpoint without removing it from the list.

on *n* Enables a breakpoint that was turned off.

out *n* Removes a breakpoint from the list.

Conditional breakpoints are used in a manner similar to data breakpoints, except that the user specifies a place in the program at which *sdb* should stop to check the data values. For example,

> mysub:99 b if xyz = 123

The above example instructs *sdb* to check the value of xyz every time the program arrives at line 99 of subroutine *mysub*. If the condition is true, then execution stops there, as with a normal breakpoint. This type of breakpoint does not monitor the value xyz at every line of code, as the data breakpoint does.

FILES

a.out
core

SEE ALSO

cc(1), sh(1), a.out(4), core(4).

WARNINGS

When *sdb* prints the value of an external variable for which there is no debugging information, a warning is printed before the value. The value is assumed to be **int** (integer).

Data which are stored in text sections are indistinguishable from functions.

Line number information in optimized functions is unreliable, and some information may be missing.

BUGS

If a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a core image.

When setting a breakpoint at a procedure, *sdb* will inconsistently produce the incorrect line number. This seems to occur when the object file is newer than the source file. Recompiling the source program will correct this problem.

NAME

sdiff – side-by-side difference program

SYNOPSIS

sdiff [options ...] file1 file2

DESCRIPTION

Sdiff uses the output of *diff*(1) to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files is printed with a blank gutter between them if the lines are identical, a < in the gutter if the line only exists in *file1*, a > in the gutter if the line only exists in *file2*, and a | for lines that are different.

For example:

```
x       |       y
a               a
b       <
c       <
d               d
        >       c
```

The following options exist:

–w *n* Use the next argument, *n*, as the width of the output line. The default line length is 130 characters.

–l Only print the left side of any lines that are identical.

–s Do not print identical lines.

–o *output* Use the next argument, *output*, as the name of a third file that is created as a user controlled merging of *file1* and *file2*. Identical lines of *file1* and *file2* are copied to *output*. Sets of differences, as produced by *diff*(1), are printed; where a set of differences share a common gutter character. After printing each set of differences, *sdiff* prompts the user with a % and waits for one of the following user-typed commands:

l append the left column to the output file

r append the right column to the output file

s turn on silent mode; do not print identical lines

v turn off silent mode

e l call the editor with the left column

e r call the editor with the right column

e b call the editor with the concatenation of left and right

e call the editor with a zero length file

q exit from the program

On exit from the editor, the resulting file is concatenated on the end of the *output* file.

SEE ALSO

diff(1), ed(1).

NAME

sed – stream editor

SYNOPSIS

sed [**–n**] [**–e** script] [**–f** sfile] [files]

DESCRIPTION

Sed copies the named *files* (standard input default) to the standard output, edited according to a script of commands. The **–f** option causes the script to be taken from file *sfile*; these options accumulate. If there is just one **–e** option and no **–f** options, the flag **–e** may be omitted. The **–n** option suppresses the default output. A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation, *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a **D** command), applies in sequence all commands whose *addresses* select that pattern space, and at the end of the script copies the pattern space to the standard output (except under **–n**) and deletes the pattern space.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval.

An *address* is either a decimal number that counts input lines cumulatively across files, a $ that addresses the last line of input, or a context address, i.e., a */regular expression/* in the style of *ed*(1) modified thus:

In a context address, the construction *\?regular expression?*, where *?* is any character, is identical to */regular expression/*. Note that in the context address **\xabc\xdefx**, the second **x** stands for itself, so that the regular expression is **abcxdef**.

The escape sequence **\n** matches a new-line *embedded* in the pattern space.

A period **.** matches any character except the *terminal* new-line of the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function **!** (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

The *text* argument consists of one or more lines, all but the last of which end with \ to hide the new-line. Backslashes in text are treated like backslashes in the replacement string of an **s** command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line. The *rfile* or *wfile* argument must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 10 distinct *wfile* arguments.

(1) **a**
text Append. Place *text* on the output before reading the next input line.

(2) **b** *label* Branch to the **:** command bearing the *label*. If *label* is empty, branch to the end of the script.

(2) **c**
text Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.

(2) **d** Delete the pattern space. Start the next cycle.

(2) **D** Delete the initial segment of the pattern space through the first new-line. Start the next cycle.

(2) **g** Replace the contents of the pattern space by the contents of the hold space.

(2) **G** Append the contents of the hold space to the pattern space.

(2) **h** Replace the contents of the hold space by the contents of the pattern space.

(2) **H** Append the contents of the pattern space to the hold space.

(1) **i**
text Insert. Place *text* on the standard output.

(2) **l** List the pattern space on the standard output in an unambiguous form. Non-printing characters are spelled in two-digit ASCII and long lines are folded.

(2) **n** Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2) **N** Append the next line of input to the pattern space with an embedded new-line. (The current line number changes.)

(2) **p** Print. Copy the pattern space to the standard output.

(2) **P** Copy the initial segment of the pattern space through the first new-line to the standard output.

(1) **q** Quit. Branch to the end of the script. Do not start a new cycle.

(2) **r** *rfile* Read the contents of *rfile*. Place them on the output before reading the next input line.

(2) **s**/*regular expression*/*replacement*/*flags*
Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character may be used instead of /. For a fuller description see *ed*(1). *Flags* is zero or more of:

g Global. Substitute for all nonoverlapping instances of the *regular expression* rather than just the first one.

p Print the pattern space if a replacement was made.

w *wfile*
Write. Append the pattern space to *wfile* if a replacement was made.

(2) **t** *label* Test. Branch to the **:** command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a **t**. If *label* is empty, branch to the end of the script.

(2) **w** *wfile*
Write. Append the pattern space to *wfile*.

(2) **x** Exchange the contents of the pattern and hold spaces.

(2) **y**/*string1*/*string2*/
Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal.

(2)**!** *function*
Don't. Apply the *function* (or group, if *function* is { } only to lines *not* selected by the address(es).

(0) **:** *label* This command does nothing; it bears a *label* for **b** and **t** commands to branch to.

(1) **=** Place the current line number on the standard output as a line.

(2) **{** Execute the following commands through a matching } only when the pattern space is selected.

(0) An empty command is ignored.

SEE ALSO

awk(1), ed(1), grep(1).

NAME

setprint – send a different page length/width to an LP line printer

SYNOPSIS

setprint lines cols

DESCRIPTION

Lp uses a default page length (66 lines) and page width (132 columns) for printing. If the file to be printed has more than 132 columns, all characters beyond 132 would either be truncated or the printer would continue to print them all on the last character position.

Setprint allows you to change the line and column size parameters to whatever your printer can handle. However, *setprint* can only be used with a parallel line printer, and that printer must be online. Otherwise an I/O error will occur.

EXAMPLE

To change the page width to to 150 columns, use *setprint* as follows:

```
setprint 66 150
```

Use the following format to set the page width back to 132 columns:

```
setprint 66 132
```

NAME

sh, rsh – shell, the standard/restricted command programming language

SYNOPSIS

sh [**–ceiknrstuvx**] [args]
rsh [**–ceiknrstuvx**] [args]

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. *Rsh* is a restricted version of the standard command interpreter *sh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See *Invocation* below 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 (octal) 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 **|** (or, for historical compatibility, 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 **&**. Of these four symbols, **;** and **&** have equal precedence, which is lower than that of **&&** and **| |**. The symbols **&&** and **| |** also have equal precedence. A semicolon (**;**) causes sequential execution of the preceding pipeline; an ampersand (**&**) causes asynchronous execution of the preceding pipeline (i.e., the shell does *not* wait for that pipeline to finish). The symbol **&&** (**| |**) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of new-lines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a simple-command or one of the following. Unless otherwise stated, 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* taken from the **in** *word* list. If **in** *word* . . . is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see *Parameter Substitution* below). 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 (see *File Name Generation* below).

if *list* **then** *list* [**elif** *list* **then** *list*] . . . [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns a zero exit status.

while *list* **do** *list* **done**
A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

(*list*)
Execute *list* in a sub-shell.

{*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 esac for while until do done { }

Comments.
A word beginning with # causes that word and all the following characters up to a new-line to be ignored.

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 new-lines 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*] . . .

Pattern-matching is not performed on *value*.

${*parameter*}
A *parameter* is a sequence of letters, digits, or underscores (a *name*), a digit, or any of the characters *, @, #, ?, –, $, and !. 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. A *name* must begin with a letter or underscore. 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). Parameter **$0** is set from argument zero when the shell is invoked.

${*parameter*:-*word*}
If *parameter* is set and is non-null then substitute its value; otherwise substitute *word*.

${*parameter*:=*word*}
If *parameter* is not set or is null 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 and is non-null then substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, then the message "parameter null or not set" is printed.

${*parameter*:+*word*}
If *parameter* is set and is non-null 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, in the following example, **pwd** is executed only if **d** is not set or is null:

```
echo ${d:- `pwd` }
```

If the colon (:) is omitted from the above expressions, then the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

#	The number of positional parameters in decimal.
-	Flags supplied to the shell on invocation or by the **set** command.
?	The decimal value returned by the last synchronously executed command.
$	The process number of this shell.
!	The process number of the last background command invoked.

The following parameters are used by the shell:

HOME	The default argument (home directory) for the *cd* command.
PATH	The search path for commands (see *Execution* below). The user may not change **PATH** if executing under *rsh*.
CDPATH	The search path for the *cd* command.
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 **new-line**.

The shell gives default values to **PATH**, **PS1**, **PS2**, and **IFS**, while **HOME** and **MAIL** are not set at all by the shell (although **HOME** *is* set by *login*(1M)).

Blank Interpretation.

After parameter and command substitution, the 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 **/**, as well as the character **/** itself, must be matched explicitly.

*	Matches any string, including the null string.
?	Matches any single character.
[. . .]	Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening ``[`` is a **"!"** then any character not enclosed is matched.

Quoting.

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

; & ( ) | ^ < > new-line space tab

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a ****. The pair **\new-line** is ignored. All characters enclosed between a pair of single quote marks (**''**), except a single quote, are quoted. Inside double quote marks (**""**), 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 new-line is typed and further input is needed to complete a command, then the secondary prompt (i.e., the value of **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 to it (by first seeking to the end-of-file); otherwise, the file is created.

<<[-]word The shell input is read up to a line that is the same as *word*, or to an 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, (unescaped) **\new-line** is ignored, and \ must be used to quote the characters \, $, `, and the first character of *word*. If - is appended to <<, then all leading tabs are stripped from *word* and from the document.

<&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 that is 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.

Redirection of output is not allowed in the restricted shell.

Environment.

The *environment* (see *environ*(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. 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:

TERM=450 cmd args and
(export TERM; TERM=450; cmd args)

are equivalent (as far as the above execution of *cmd* is concerned).

If the **-k** flag is set, *all* keyword arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and then **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, with the exception of signal 11 (but see also the **trap** command below).

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 *exec*(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is **:/bin:/usr/bin** (specifying the current directory, **/bin**, and **/usr/bin**, in that order). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a / then the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a sub-shell.

Special Commands.

The following commands are executed in the shell process and, except as specified, no input/output redirection is permitted for such commands:

: No effect; the command does nothing. A zero exit code is returned.

. *file* Read and execute commands from *file* and return. The search path specified by **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*. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (**:**). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / then the search path is not used. Otherwise, each directory in the path is searched for *arg*. The *cd* command may not be executed by *rsh*.

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 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 cause the shell to exit.)

export [*name* ...]
The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, then a list of all names that are exported in this shell is printed.

newgrp [*arg* ...]
Equivalent to **exec newgrp** *arg* ....

read [*name* ...]
One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. The return code is 0 unless an end-of-file is encountered.

readonly [*name* ...]
The given *name*s are marked *readonly* and the values of these *name*s may not be changed by subsequent assignment. If no arguments are given, then a list of all *readonly* names is printed.

set [**--ekntuvx** [*arg* ...]]

-e Exit immediately if a command exits with a non-zero exit status.

-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.
-- Do not change any of the flags; useful in setting **$1** to -.

Using + rather than - causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in $-. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given then the values of all names are printed.

shift [*n*]
The positional parameters from **$n+1** ... are renamed **$1** .... If *n* is not given, it is assumed to be 1.

test
Evaluate conditional expressions. See *test*(1) for usage and description.

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. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. 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 the commands it invokes. If *n* is 0 then the command *arg* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

ulimit [**-fp**] [*n*]
imposes a size limit of *n*
-f imposes a size limit of *n* blocks on files written by child processes (files of any size may be read). With no argument, the current limit is printed.
-p changes the pipe size to *n* (UNIX/RT only).
If no option is given, **-f** is assumed.

umask [*nnn*]
The user file-creation mask is set to *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 and the return code is zero.

Invocation.

If the shell is invoked through *exec*(2) and the first character of argument zero is −, commands are initially read from **/etc/profile** and then from **$HOME/.profile**, if such files exist. Thereafter, commands are read as described below, which is also the case when the shell is invoked as **/bin/sh**. The flags below are interpreted by the shell on invocation only; Note that unless the **−c** or **−s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

−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. Any remaining arguments specify the positional parameters. 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, then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.

−r If the **−r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

Rsh Only.

Rsh is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of *rsh* are identical to those of *sh*, except that the following are disallowed:

changing directory (see *cd*(1)),
setting the value of **$PATH,**
specifying path or command names containing /,
redirecting output (> and >>).

The restrictions above are enforced after **.profile** is interpreted.

When a command to be executed is found to be a shell procedure, *rsh* invokes *sh* to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the **.profile** has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (i.e., **/usr/rbin**) that can be safely invoked by *rsh*. Some systems also provide a restricted editor *red*.

EXIT STATUS

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 the **exit** command above).

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

cd(1), env(1), login(1M), newgrp(1), test(1), umask(1), dup(2), exec(2), fork(2), pipe(2), signal(2), ulimit(2), umask(2), wait(2), a.out(4), profile(4), environ(5).

BUGS

The command **readonly** (without arguments) produces the same output as the command **export**.

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 that file by another name.

NAME

shform – displays menus and forms and returns user input to Bourne Shell procedures.

SYNOPSIS

RET = 'shform [-u] formname'

DESCRIPTION

The *shform* process displays a menu or form, waits for user input, and returns the result to the shell procedure.

Formname is a text document, called a form description file, that describes the menu or form to be displayed. Entries in the file use a keyword = value syntax. (The form and menu keywords are described below.) The file must be located in the */usr/lib/ua* directory. To insert a comment in the file, start the line with a pound sign (#).

The value returned by the file is stored in the shell variable *RET* as a list of words separated by spaces. If an error occurs, then $? will contain an error code.

-u causes *shform* to place its menu or form in the current window, resizing it appropriately to fix the menu or form. This option is recommended.

Shform returns the following exit codes:

0 – AOK
1 – Argument error
2 – Out of memory (malloc failed)
3 – Internal table overflow
4 – Syntax error in form description file

The words in $*RET* are as follows:

word 1 = Name of terminating key

if form, words 2 – n = Values of the form's fields

if menu, word 2 = Name of selected menu item

if multiselect menu, words 2 – n = Name of selected menu items

Form Definition Keywords

Form = form name

Flags the start of a form. It is followed by a series of field definitions. The form name specified here is used as the title of the form. Only one Form keyword can be used in the file.

Name = field name

Follows a Form keyword and starts a field definition. The field name specified here is used as the prompt for the field. The field name definition is followed by field attribute definitions:

Prompt = prompt string

Displayed on the prompt line when the field is the current field.

Frow = number
Defines the row in the form where the current field displays.

Ncol = number.
Defines the column in the form where the field name displays.

Fcol = number
Defines the column in the form where the field value displays.

Flen = number
Defines the maximum length of the field value, in columns.

Value = initial field value
Defines the initial contents (default value) for the field.

Rmenu = menu name
If the field has an associated menu of options, this keyword is included. The menu name must be defined later in the file with the Menu keyword (see below).

Menuonly
If this keyword is present then user editing of the field is forbidden and any key which is typed will cause the associated menu to display.

Menu Definition Keywords

Menu = menu name
Begins a menu definition. When no form is defined, *shform* displays a menu instead of a form. In this case, only the first defined menu is displayed. If a form is defined, then the only menus displayed are those referenced in the form fields (via the Rmenu keyword, defined above). The Menu keyword is followed by a series of menu attribute definitions.

Prompt = prompt string
The prompt string is displayed on the prompt line when the menu is displayed.

Rows = number
Defines the number of rows in the menu display.

Columns = number
Defines the number of columns in the menu display. If neither Rows nor Columns is defined, then the built-in menu heuristic is used for determining the number of rows and columns in the menu.

Multiple
If this keyword is present, the menu is multi-select. Otherwise, the menu is a single select menu.

Name = item name
Follows the menu attributes and specifies the name displayed in the menu. This keyword is returned to the caller when this item is selected.

SEE ALSO

menu(3T), form(3T), tam(3T).

NAME

size – print section sizes of common object files

SYNOPSIS

size [**–o**] [**–x**] [**–V**] files

DESCRIPTION

The *size* command produces section size information for each section in the common object files. The size of the text, data, bss (uninitialized data), and shared library sections are printed along with the total size of the object file. If an archive file is input to the *size* command the information for all archive members is displayed.

Numbers will be printed in decimal unless either the **–o** or the **–x** option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **–V** flag will supply the version information on the *size* command.

SEE ALSO

as(1), cc(1), ld(1), a.out(4), ar(4).

DIAGNOSTICS

size: name: cannot open
if *name* cannot be read.

size: name: bad magic
if *name* is not an appropriate common object file.

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(3C).

BUGS

Time must be less than 2147483647 seconds.

NAME

sort – sort and/or merge files

SYNOPSIS

sort [–**cmubdfinrtx**] [+pos1 [–pos2]] **...** [–**o** output] [names]

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 non-numeric 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 non-blank 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 non-empty non-blank 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.

m Merge only, the input files are already sorted.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

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.

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*(4)) sorted by user ID (the third 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???

SEE ALSO

comm(1), join(1), uniq(1).

DIAGNOSTICS

Comments and exits with non-zero status for various trouble conditions and for disorder discovered under option **-c**.

BUGS

Very long lines are silently truncated.

NAME

spell, hashmake, spellin, hashcheck – find spelling errors

SYNOPSIS

spell [**-v**] [**-b**] [**-x**] [**-l**] [+local_file] [files]

/usr/lib/spell/hashmake

/usr/lib/spell/spellin n

/usr/lib/spell/hashcheck spelling_list

DESCRIPTION

Spell collects words from the named *files* and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no *files* are named, words are collected from the standard input.

Spell ignores most *nroff*(1), *tbl*(1), and *eqn*(1) constructions.

Under the **-v** option, all words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.

Under the **-b** option, British spelling is checked. Besides preferring *centre*, *colour*, *programme*, *speciality*, *traveled*, etc., this option insists upon *-ise* in words like *standardise*, Fowler and the OED to the contrary notwithstanding.

Under the **-x** option, every plausible stem is printed with = for each word.

By default, *spell* (like *deroff*(1)) follows chains of included files (**.so** and **.nx** *troff* requests), *unless* the names of such included files begin with **/usr/lib**. Under the **-l** option, *spell* will follow the chains of *all* included files.

Under the *+local_file* option, words found in *local_file* are removed from *spell*'s output. *Local_file* is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to *spell*'s own spelling list) for each job.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective with respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine, and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings (see *FILES*). Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g., thier=thy-y+ier) that would otherwise pass.

Three routines help maintain and check the hash lists used by *spell*:

hashmake Reads a list of words from the standard input and writes the corresponding nine-digit hash code on the

standard output.

spellin Reads *n* hash codes from the standard input and writes a compressed spelling list on the standard output.

hashcheck Reads a compressed *spelling_list* and recreates the nine-digit hash codes for all the words in it; it writes these codes on the standard output.

FILES

D_SPELL=/usr/lib/spell/hlist[ab]	hashed spelling lists, American & British
S_SPELL=/usr/lib/spell/hstop	hashed stop list
H_SPELL=/usr/lib/spell/spellhist	history file
/usr/lib/spell/spellprog	program

SEE ALSO

deroff(1), eqn(1), sed(1), sort(1), tbl(1), tee(1).

BUGS

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions; typically, these are kept in a separate local file that is added to the hashed *spelling_list* via *spellin*.
The British spelling feature was done by an American.

NAME

split – split a file into pieces

SYNOPSIS

split [*–n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000 lines) onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically, up to **zz** (a maximum of 676 files). *Name* cannot be longer than 12 characters. If no output name is given, **x** is default.

If no input file is given, or if – is given in its stead, then the standard input file is used.

SEE ALSO

bfs(1), csplit(1).

NAME

strip – strip symbol and line number information from a common object file

SYNOPSIS

strip [**–l**] [**–x**] [**–r**] [**–s**] [**–V**] file-names

DESCRIPTION

The *strip* command strips the symbol table and line number information from common object files, including archives. Once this has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

–l Strip line number information only; do not strip any symbol table information.

–x Do not strip static or external symbol information.

–r Reset the relocation indexes into the symbol table.

–s Reset the line number indexes into the symbol table (do not remove). reset the relocation indexes into the symbol table.

–V Version of strip command executing.

If there are any relocation entries in the object file and any symbol table information is to be stripped, *strip* will complain and terminate without stripping *file-name* unless the **–r** flag is used.

If the *strip* command is executed on a common archive file (see *ar*(4)) the archive symbol table will be removed. The archive symbol table must be restored by executing the *ar*(1) command with the **s** option before the archive can be link edited by the *ld*(1) command. *Strip*(1) will instruct the user with appropriate warning messages when this situation arises.

The purpose of this command is to reduce the file storage overhead taken by the object file.

FILES

/usr/tmp/strp??????

SEE ALSO

as(1), cc(1), ld(1), ar(4), a.out(4).

DIAGNOSTICS

strip: name: cannot open
if *name* cannot be read.

strip: name: bad magic
if *name* is not an appropriate common object file.

strip: name: relocation entries present; cannot strip
if *name* contains relocation entries and the **–r** flag is not used, the symbol table information cannot be stripped.

NAME

stty – set the options for a terminal

SYNOPSIS

stty [**–a**] [**–g**] [options]

DESCRIPTION

Stty sets certain terminal I/O options for the device that is the current standard input; without arguments, it reports the settings of certain options; with the **–a** option, it reports all of the option settings; with the **–g** option, it reports current settings in a form that can be used as an argument to another *stty* command. Detailed information about the modes listed in the first five groups below may be found in *termio*(7) for asynchronous lines. Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sanity checking is performed. The options are selected from the following:

Control Modes

parenb (**–parenb**)	enable (disable) parity generation and detection.
parodd (**–parodd**)	select odd (even) parity.
cs5 cs6 cs7 cs8	select character size (see *termio*(7)).
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. (All speeds are not supported by all hardware interfaces.)
hupcl (**–hupcl**)	hang up (do not hang up) DATA-PHONE connection on last close.
hup (**–hup**)	same as **hupcl** (**–hupcl**).
cstopb (**–cstopb**)	use two (one) stop bits per character.
cread (**–cread**)	enable (disable) the receiver.
clocal (**–clocal**)	assume a line without (with) modem control.

Input Modes

ignbrk (**–ignbrk**)	ignore (do not ignore) break on input.
brkint (**–brkint**)	signal (do not signal) INTR on break.
ignpar (**–ignpar**)	ignore (do not ignore) parity errors.
parmrk (**–parmrk**)	mark (do not mark) parity errors (see *termio*(7)).
inpck (**–inpck**)	enable (disable) input parity checking.
istrip (**–istrip**)	strip (do not strip) input characters to seven bits.
inlcr (**–inlcr**)	map (do not map) NL to CR on input.
igncr (**–igncr**)	ignore (do not ignore) CR on input.
icrnl (**–icrnl**)	map (do not map) CR to NL on input.
iuclc (**–iuclc**)	map (do not map) upper-case alphabetics to lower case on input.

ixon (–ixon) enable (disable) START/STOP output control. Output is stopped by sending an ASCII DC3 and started by sending an ASCII DC1.

ixany (–ixany) allow any character (only DC1) to restart output.

ixoff (–ixoff) request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

Output Modes

opost (–opost) post-process output (do not post-process output; ignore all other output modes).

olcuc (–olcuc) map (do not map) lower-case alphabetics to upper case on output.

onlcr (–onlcr) map (do not map) NL to CR-NL on output.

ocrnl (–ocrnl) map (do not map) CR to NL on output.

onocr (–onocr) do not (do) output CRs at column zero.

onlret (–onlret) on the terminal NL performs (does not perform) the CR function.

ofill (–ofill) use fill characters (use timing) for delays.

ofdel (–ofdel) fill characters are DELs (NULs).

cr0 cr1 cr2 cr3 select style of delay for carriage returns (see *termio*(7)).

nl0 nl1 select style of delay for line-feeds (see *termio*(7)).

tab0 tab1 tab2 tab3 select style of delay for horizontal tabs (see *termio*(7).

bs0 bs1 select style of delay for backspaces (see *termio*(7)).

ff0 ff1 select style of delay for form-feeds (see *termio*(7)).

vt0 vt1 select style of delay for vertical tabs (see *termio*(7)).

Local Modes

isig (–isig) enable (disable) the checking of characters against the special control characters INTR and QUIT.

icanon (–icanon) enable (disable) canonical input (ERASE and KILL processing).

xcase (–xcase) canonical (unprocessed) upper/lower-case presentation.

echo (–echo) echo back (do not echo back) every character typed.

echoe (–echoe) echo (do not echo) ERASE character as a backspace-space-backspace string. Note: this mode will erase the ERASEed character on many CRT terminals; however, it does *not* keep track of column position and, as a result, may be confusing on escaped characters, tabs, and backspaces.

echok (–echok) echo (do not echo) NL after KILL character.

lfkc (**−lfkc**) the same as **echok** (**−echok**); obsolete.
echonl (**−echonl**) echo (do not echo) NL.
noflsh (**−noflsh**) disable (enable) flush after INTR or QUIT.
stwrap (**−stwrap**) disable (enable) truncation of lines longer than 79 characters on a synchronous line.
stflush (**−stflush**) enable (disable) flush on a synchronous line after every *write*(2).
stappl (**−stappl**) use application mode (use line mode) on a synchronous line.

Control Assignments

control-character c set *control-character* to *c*, where *control-character* is **erase**, **kill**, **intr**, **quit**, **eof**, **eol**, **ctab**, **min**, or **time** (**ctab** is used with **−stappl**; see *termio*(7)). If *c* is preceded by an (escaped from the shell) caret (^), then the value used is the corresponding CTRL character (e.g., "^**d**" is a **CTRL-d**); "^**?**" is interpreted as DEL and "^−" is interpreted as undefined.
line *i* set line discipline to *i* ($0 < i < 127$).

Combination Modes

evenp or **parity** enable **parenb** and **cs7**.
oddp enable **parenb**, **cs7**, and **parodd**.
−parity, **−evenp**, or **−oddp** disable **parenb**, and set **cs8**.
raw (**−raw** or **cooked**) enable (disable) raw input and output (no ERASE, KILL, INTR, QUIT, EOT, or output post processing).
nl (**−nl**) unset (set) **icrnl**, **onlcr**. In addition **−nl** unsets **inlcr**, **igncr**, **ocrnl**, and **onlret**.
lcase (**−lcase**) set (unset) **xcase**, **iuclc**, and **olcuc**.
LCASE (**−LCASE**) same as **lcase** (**−lcase**).
tabs (**−tabs** or **tab3**) preserve (expand to spaces) tabs when printing.
ek reset ERASE and KILL characters back to normal # and @.
sane resets all modes to some reasonable values.
term set all modes suitable for the terminal type *term*, where *term* is one of **tty33**, **tty37**, **vt05**, **tn300**, **ti700**, or **tek**.

SEE ALSO

tabs(1), ioctl(2).
termio(7) in the *UNIX System Administrator's Manual*.

NAME

su – become super-user or another user

SYNOPSIS

su [–] [name [arg . . .]]

DESCRIPTION

Su allows one to become another user without logging off. The default user *name* is **root** (i.e., super-user).

To use *su*, the appropriate password must be supplied (unless one is already super-user). If the password is correct, *su* will execute a new shell with the user ID set to that of the specified user. To restore normal user ID privileges, type an **EOF** to the new shell.

Any additional arguments are passed to the shell, permitting the super-user to run shell procedures with restricted privileges (an *arg* of the form **–c** *string* executes *string* via the shell). When additional arguments are passed, **/bin/sh** is always used. When no additional arguments are passed, *su* uses the shell specified in the password file.

An initial – flag causes the environment to be changed to the one that would be expected if the user actually logged in again. This is done by invoking the shell with an *arg0* of **–su** causing the **.profile** in the home directory of the new user ID to be executed. Otherwise, the environment is passed along with the possible exception of **$PATH**, which is set to **/bin:/etc:/usr/bin:/usr/local/bin** for root. Note that the **.profile** can check *arg0* for **–sh** or **–su** to determine how it was invoked.

FILES

/etc/passwd	system's password file
$HOME/.profile	user's profile

SEE ALSO

env(1), login(1M), sh(1), environ(5).

NAME

sum – print checksum and block count of a file

SYNOPSIS

sum [**–r**] file

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. It is typically used to look for bad spots, or to validate a file communicated over some transmission line. The option **–r** causes an alternate algorithm to be used in computing the checksum.

SEE ALSO

wc(1).

DIAGNOSTICS

"Read error" is indistinguishable from end of file on most devices; check the block count.

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. It will flush all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point will be saved. See *sync*(2) for details.

SEE ALSO

sync(2).

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [tabspec] [**+mn**] [**–Ttype**]

DESCRIPTION

Tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user must of course be logged in on a terminal with remotely-settable hardware tabs.

Users of GE TermiNet terminals should be aware that they behave in a different way than most other terminals for some tab settings: the first number in a list of tab settings becomes the *left margin* on a TermiNet terminal. Thus, any list of tab numbers whose first element is other than 1 causes a margin to be left on a TermiNet, but not on other terminals. A tab list beginning with 1 causes the same effect regardless of terminal type. It is possible to set a left margin on some other terminals, although in a different way (see below).

Four types of tab specification are accepted for *tabspec*: "canned," repetitive, arbitrary, and file. If no *tabspec* is given, the default value is **–8**, i.e., UNIX "standard" tabs. The lowest column number is 1. Note that for *tabs*, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0, e.g., the DASI 300, DASI 300s, and DASI 450.

–code Gives the name of one of a set of "canned" tabs. The legal codes and their meanings are as follows:

–a 1,10,16,36,72
Assembler, IBM S/370, first format

–a2 1,10,16,40,72
Assembler, IBM S/370, second format

–c 1,8,12,16,20,55
COBOL, normal format

–c2 1,6,10,14,49
COBOL compact format (columns 1-6 omitted). Using this code, the first typed character corresponds to card column 7, one space gets you to column 8, and a tab reaches column 12. Files using this tab setup should include a format specification as follows:

<:t–c2 m6 s66 d:>

–c3 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1-6 omitted), with more tabs than **–c2.** This is the recommended format for COBOL. The appropriate format specification is:

<:t–c3 m6 s66 d:>

–f 1,7,11,15,19,23
FORTRAN

–p 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

–s 1,10,55
SNOBOL

−u 1,12,20,44
UNIVAC 1100 Assembler

In addition to these "canned" formats, three other types exist:

−n A repetitive specification requests tabs at columns 1+*n*, 1+2**n*, etc. Note that such a setting leaves a left margin of *n* columns on TermiNet terminals *only*. Of particular importance is the value **−8**: this represents the UNIX "standard" tab setting, and is the most likely tab setting to be found at a terminal. It is required for use with the *nroff* **−h** option for high-speed output. Another special case is the value **−0**, implying no tabs at all.

n1,n2,...
The arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered identical.

−−file If the name of a file is given, *tabs* reads the first line of the file, searching for a format specification. If it finds one there, it sets the tab stops according to it, otherwise it sets them as **−8**. This type of specification may be used to make sure that a tabbed file is printed with correct tab settings, and would be used with the *pr*(1) command:

tabs −− file; pr file

Any of the following may be used also; if a given flag occurs more than once, the last value given takes effect:

−T*type* *Tabs* usually needs to know the type of terminal in order to set tabs and always needs to know the type to set margins. *Type* is a name listed in *term*(5). If no **−T** flag is supplied, *tabs* searches for the **$TERM** value in the *environment* (see *environ*(5)). If no *type* can be found, *tabs* tries a sequence that will work for many terminals.

+m*n* The margin argument may be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n+1* the left margin. If **+m** is given without a value of *n*, the value assumed is 10. For a TermiNet, the first value in the tab list should be 1, or the margin will move even further to the right. The normal (leftmost) margin on most terminals is obtained by **+m0**. The margin for most terminals is reset only when the **+m** flag is given explicitly.

Tab and margin setting is performed via the standard output.

DIAGNOSTICS

illegal tabs	when arbitrary tabs are ordered incorrectly.
illegal increment	when a zero or missing increment is found in an arbitrary specification.

unknown tab code	when a "canned" code cannot be found.
can't open	if *--file* option used, and file can't be opened.
file indirection	if *--file* option used and the specification in that file points to yet another file. Indirection of this form is not permitted.

SEE ALSO

nroff(1), environ(5), term(5).

BUGS

There is no consistency among different terminals regarding ways of clearing tabs and setting the left margin.

It is generally impossible to usefully change the left margin without also setting tabs.

Tabs clears only 20 tabs (on terminals requiring a long sequence), but is willing to set 40.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [±[number][**lbc**[**f**]]] [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 (if *number* is null, the value 10 is assumed). *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.

With the **–f** ("follow") option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop, wherein it sleeps for a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail –f fred
```

will print the last ten lines of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed. As another example, the command:

```
tail –15cf fred
```

will print the last 15 characters of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed.

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 file archiver

SYNOPSIS

tar [key] [files]

DESCRIPTION

Tar saves and restores files on magnetic 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 *files* (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 function.

x The named *files* are extracted from the tape. If a 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 *files* argument is given, the entire content of the tape is extracted. Note that if several files with the same name are on the tape, the last one overwrites all earlier ones.

t The names of the specified files are listed each time that they occur on the tape. If no *files* argument is given, all the names on the tape are listed.

u The named *files* are added to the tape if they are not already there, or have been modified since last written on that tape.

c Create a new tape; writing begins at the beginning of the tape, instead of after the last file. This command implies the **r** function.

The following characters may be used in addition to the letter that selects the desired function:

0,...,7 This modifier selects the drive on which the tape is mounted. The default is **1**.

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 the name of the file, and then wait for the user's confirmation. If a word beginning with **y** is given, the action is performed. Any other input means "no".

f causes *tar* to use the next argument as the name of the archive instead of **/dev/mt?**. If the name of the file is **–**, *tar* writes to the standard output or reads from the standard input, whichever is appropriate. Thus, *tar* can be used as the head or tail of a pipeline. *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 is 1, the maximum is 20. 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 being dumped. If **l** is not specified, no error messages are printed.

m tells *tar* not to restore the modification times. The modification time of the file will be the time of extraction.

FILES

/dev/mt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if not enough memory is available to hold the link tables.

BUGS

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. The current magnetic tape driver cannot backspace raw magnetic tape. If the archive is on a disk file, the **b** option should not be used at all, because updating an archive stored on disk can destroy it.
The current limit on file-name length is 100 characters.

NAME

tbl – format tables for nroff or troff

SYNOPSIS

tbl [**–TX**] [files]

DESCRIPTION

Tbl is a preprocessor that formats tables for *nroff* or *troff* (not included on the UNIX PC). The input files are copied to the standard output, except for lines between **.TS** and **.TE** command lines, which are assumed to describe tables and are re-formatted by *tbl*. (The **.TS** and **.TE** command lines are not altered by *tbl*).

.TS is followed by global options. The available global options are:

center	center the table (default is left-adjust);
expand	make the table as wide as the current line length;
box	enclose the table in a box;
doublebox	enclose the table in a double box;
allbox	enclose each item of the table in a box;
tab (*x*)	use the character *x* instead of a tab to separate items in a line of input data.

The global options, if any, are terminated with a semi-colon (**;**).

Next come lines describing the format of each line of the table. Each such format line describes one line of the actual table, except that the last format line (which must end with a period) describes *all* remaining lines of the table. Each column of each line of the table is described by a single key-letter, optionally followed by specifiers that determine the font and point size of the corresponding item, that indicate where vertical bars are to appear between columns, that determine column width, inter-column spacing, etc. The available key-letters are:

c	center item within the column;
r	right-adjust item within the column;
l	left-adjust item within the column;
n	numerically adjust item in the column: units positions of numbers are aligned vertically;
s	span previous item on the left into this column;
a	center longest line in this column and then left-adjust all other lines in this column with respect to that centered line;
^	span down previous entry in this column;
_	replace this entry with a horizontal line;
=	replace this entry with a double horizontal line.

The characters **B** and **I** stand for the bold and italic fonts, respectively; the character **|** indicates a vertical line between columns.

The format lines are followed by lines containing the actual data for the table, followed finally by **.TE**. Within such data lines, data items are normally separated by tab characters.

If a data line consists of only _ or =, a single or double line, respectively, is drawn across the table at that point; if a *single item* in a data line consists of only _ or =, then that item is replaced by a single or double line.

Full details of all these and other features of *tbl* are given in the reference manual cited below.

The **-TX** option forces *tbl* to use only full vertical line motions, making the output more suitable for devices that cannot generate partial vertical line motions (e.g., line printers).

If no file names are given as arguments (or if - is specified as the last argument), *tbl* reads the standard input, so it may be used as a filter. When it is used with *eqn*(1) or *neqn*, *tbl* should come first to minimize the volume of data passed through pipes.

EXAMPLE

If we let → represent a tab (which should be typed as a genuine tab), then the input:

```
.TS
center box ;
cB s s
cI | cI s
^ | c c
l | n n .
Household Population
_
Town→Households
→Number→Size
=
Bedminster→789→3.26
Bernards Twp.→3087→3.74
Bernardsville→2018→3.30
Bound Brook→3425→3.04
Bridgewater→7897→3.81
Far Hills→240→3.19
.TE
```

yields:

Household Population		
Town	*Households*	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Bridgewater	7897	3.81
Far Hills	240	3.19

SEE ALSO

TBL–A Program to Format Tables in the *UNIX System Document Processing Guide*.
cw(1), eqn(1), mm(1), nroff(1), mm(5).

BUGS

See *BUGS* under *nroff*(1).

NAME

tc – phototypesetter simulator

SYNOPSIS

tc [**–t**] [**–sn**] [**–pl**] [file]

DESCRIPTION

Tc interprets its input (standard input default) as device codes for a Wang Laboratories, Inc. C/A/T phototypesetter. The standard output of *tc* is intended for a Tektronix 4014 terminal with ASCII and APL character sets. The sixteen typesetter sizes are mapped into the 4014's four sizes; the entire TROFF character set is drawn using the 4014's character generator, with overstruck combinations where necessary. Typical usage is:

troff –t files | tc

At the end of each page, *tc* waits for a new-line (empty line) from the keyboard before continuing on to the next page. In this wait state, the command **e** will *suppress* the screen erase before the next page; **s***n* will cause the next *n* pages to be skipped; and **!***cmd* will send *cmd* to the shell.

The command line options are:

–t Don't wait between pages (for directing output into a file).

–s*n* Skip the first *n* pages.

–p*l* Set page length to *l*; *l* may include the scale factors **p** (points), **i** (inches), **c** (centimeters), and **P** (picas); default is picas.

SEE ALSO

4014(1), sh(1).

BUGS

Font distinctions are lost.

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*. The **–i** option ignores interrupts; the **–a** option causes the output to be appended to the *files* rather than overwriting them.

NAME

test – condition evaluation command

SYNOPSIS

test expr
[expr]

DESCRIPTION

Test evaluates the expression *expr* and, if its value is true, returns a zero (true) exit status; otherwise, a non-zero (false) exit status is returned; *test* also returns a non-zero exit status if there are no arguments. The following primitives are used to construct *expr*:

-r *file*	true if *file* exists and is readable.
-w *file*	true if *file* exists and is writable.
-x *file*	true if *file* exists and is executable.
-f *file*	true if *file* exists and is a regular file.
-d *file*	true if *file* exists and is a directory.
-c *file*	true if *file* exists and is a character special file.
-b *file*	true if *file* exists and is a block special file.
-p *file*	true if *file* exists and is a named pipe (fifo).
-u *file*	true if *file* exists and its set-user-ID bit is set.
-g *file*	true if *file* exists and its set-group-ID bit is set.
-k *file*	true if *file* exists and its sticky bit is set.
-s *file*	true if *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 non-zero.
s1 **=** *s2*	true if strings *s1* and *s2* are identical.
s1 **!=** *s2*	true if strings *s1* and *s2* are *not* identical.
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**, and **-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 (**-a** has higher precedence than **-o**).
(expr **)**	parentheses for grouping.

Notice that all the operators and flags are separate arguments to *test*. Notice also that parentheses are meaningful to the shell and, therefore, must be escaped.

SEE ALSO

find(1), sh(1).

WARNING

In the second form of the command (i.e., the one that uses [ ], rather than the word *test*), the square brackets must be delimited by blanks.

Some UNIX systems do not recognize the second form of the command.

NAME

time – time a command

SYNOPSIS

time command

DESCRIPTION

The *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 execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on standard error.

SEE ALSO

times(2).

NAME

touch – update access and modification times of a file

SYNOPSIS

touch [**–amc**] [mmddhhmm[yy]] files

DESCRIPTION

Touch causes the access and modification times of each argument to be updated. If no time is specified (see *date*(1)) the current time is used. The **–a** and **–m** options cause touch to update only the access or modification times respectively (default is **–am**). The **–c** option silently prevents *touch* from creating the file if it did not previously exist.

The return code from *touch* is the number of files for which the times could not be successfully modified (including files that did not exist and were not created).

SEE ALSO

date(1), utime(2).

NAME

tr – translate characters

SYNOPSIS

tr [**−cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. Any combination of the options **−cds** may be used:

−c Complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 001 through 377 octal.

−d Deletes all input characters in *string1*.

−s Squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[a−z] Stands for the string of characters whose ASCII codes run from character **a** to character **z**, inclusive.

[a*n] Stands for *n* repetitions of **a**. If the first digit of *n* is **0**, *n* is considered octal; otherwise, *n* is taken to be decimal. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character \ may be used as in the shell to remove special meaning from any character in a string. In addition, \ followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

The following example creates a list of all the words in *file1* one per line in *file2*, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the shell; 012 is the ASCII code for newline.

```
tr −cs "[A−Z][a−z]" "[\012*]" <file1 >file2
```

SEE ALSO

ed(1), sh(1), ascii(5).

BUGS

Won't handle ASCII **NUL** in *string1* or *string2*; always deletes **NUL** from input.

NAME

true, false – provide truth values

SYNOPSIS

true

false

DESCRIPTION

True does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1) such as:

```
while true
do
        command
done
```

The following UNIX PC files are linked to either true or false:

/bin/mc68k
/bin/pdp11
/bin/u370
/bin/u3b
/bin/vax

SEE ALSO

sh(1).

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

tset – set terminal modes

SYNOPSIS

tset [options] [**–m** [*ident*][*test baudrate*]:*type* ...] [type]

DESCRIPTION

Tset causes terminal dependent processing such as setting erase and kill characters, setting or resetting delays, and the like. It first determines the *type* of terminal involved, names for which are specified by the */etc/termcap* data base, and then does necessary initializations and mode settings. In the case where no argument types are specified, *tset* simply reads the terminal type out of the environment variable *TERM* and re-initializes the terminal. The rest of this manual concerns itself with type initialization, done typically once at login, and options used at initialization time to determine the terminal type and set up terminal modes.

When used in a startup script *.profile* it is desirable to give information about the types of terminal usually used on terminals which are not hardwired. These ports are initially identified as being *dialup* or *plugboard* or *arpanet*, etc. To specify what terminal type is usually used on these ports **–m** is followed by the appropriate port type identifier, an optional baud-rate specification, and the terminal type to be used if the mapping conditions are satisfied. If more than one mapping is specified, the first applicable mapping prevails. A missing type identifier matches all identifiers.

Baud rates are specified as with *stty*(1), and are compared with the speed of the diagnostic output (which is almost always the control terminal). The baud rate test may be any combination of: **>**, **=**, **<**, **@**, and **!**; **@** is a synonym for **=** and **!** inverts the sense of the test. To avoid problems with metacharacters, it is best to place the entire argument to **–m** within " ´ " characters.

Thus

```
tset –m ´dialup>300:adm3a´ –m dialup:dw2 –m ´plugboard:?adm3a´
```

causes the terminal type to be set to an *adm3a* if the port in use is a dialup at a speed greater than 300 baud; to a *dw2* if the port is (otherwise) a dialup (i.e. at 300 baud or less). If the *type* above begins with a question mark, the user is asked if s/he really wants that type. A null response means to use that type; otherwise, another type can be entered which will be used instead. Thus, in this case, the user will be queried on a plugboard port as to whether they are using an *adm3a*. For other ports the port type will be taken from the **/etc/ttytype** file or a final, default *type* option may be given on the command line not preceded by a **–m**.

It is often desirable to return the terminal type, as specified by the **–m** options, and information about the terminal to a shell's environment. This can be done using the **–s** option; using the Bourne shell, *sh*(1):

eval `tset -s *options...*`

These commands cause *tset* to generate as output a sequence of shell commands which place the variables *TERM* and *TERMCAP* in the environment; see *environ*(5).

Once the terminal type is known, *tset* engages in terminal mode setting. This normally involves sending an initialization sequence to the terminal and setting the single character erase (and optionally the line-kill (full line erase)) characters.

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is the default erase character ('#' on standard systems), the erase character is changed to a Control-H (backspace).

The options are:

-e set the erase character to be the named character *c* on all terminals, the default being the backspace character on the terminal, usually ^H.

-k is similar to **-e** but for the line kill character rather than the erase character; *c* defaults to ^X (for purely historical reasons); ^U is the preferred setting. No kill processing is done if **-k** is not specified.

-I suppresses outputting terminal initialization strings.

-Q suppresses printing the "Erase set to" and "Kill set to" messages.

-S (lowercase) Outputs the strings to be assigned to *TERM* and *TERMCAP* in the environment rather than commands for a shell.

FILES

/etc/ttytype	terminal id to type map database
/etc/termcap	terminal capability database

SEE ALSO

sh(1), stty(1), environ(5), ttytype(5), termcap(5)

BUGS

Should be merged with *stty*(1).

NOTES

For compatibility with earlier versions of *tset* a number of flags are accepted whose use is discouraged:

-d type equivalent to **-m** dialup:type

-p type equivalent to **-m** plugboard:type

-a type equivalent to **-m** arpanet:type

-E c Sets the erase character to *c* only if the terminal can backspace.

- prints the terminal type on the standard output

-r prints the terminal type on the diagnostic output.

NAME

tsort – topological sort

SYNOPSIS

tsort [file]

DESCRIPTION

Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1).

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

BUGS

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty – get the terminal's name

SYNOPSIS

tty [**–l**] [**–s**]

DESCRIPTION

Tty prints the path name of the user's terminal. The **–l** option prints the synchronous line number to which the user's terminal is connected, if it is on an active synchronous line. The **–s** option inhibits printing of the terminal's path name, allowing one to test just the exit code.

EXIT CODES

2	if invalid options were specified,
0	if standard input is a terminal,
1	otherwise.

DIAGNOSTICS

"not on an active synchronous line" if the standard input is not a synchronous terminal and **–l** is specified.

"not a tty" if the standard input is not a terminal and **–s** is not specified.

NAME

uahelp – user agent help process

SYNOPSIS

uahelp **–h** helpfile [**–t** title] [**–d** debugfile]

DESCRIPTION

Uahelp is a help facility which is driven by a text file (*helpfile*). The syntax of this file is described below.

Title, if specified, is the title of the initial help display.

If the **–d** (debug) option is specified, then as *helpfile* is being parsed, the lines are written to *debugfile*. When a syntax error occurs during the parsing of *helpfile*, then *uahelp* displays an error message and quits. The line containing the error is the last line written to *debugfile*. This option is used to debug new *helpfiles*.

Helpfile is an ordinary ASCII text file, with a "keyword = value" syntax. The following keywords are defined:

Keyword	**Value**
Wlabel	Window label
Contents	Lists of help displays in this file
Name	Name of current help display
Llabel	Long screen label for current display
Slabel	Short screen label for current display
Branch	List of help displays available via SLK's from the current help display
Title	Title of current help display
Text	Text of current help display

All keywords must be case correct and followed by an equal sign (=) and a value. The **Wlabel** and **Contents** keywords must be defined at the beginning of the *helpfile*, and they are followed by a series of definitions of the individual help displays, one for each display listed under **Contents**.

The individual help displays begin with a **Name** definition, which must be one of the names listed under **Contents**. The remaining keyword definitions apply to the current help display, up until the **Text** keyword, which terminates the help display definition.

The value of the **Contents** and **Branch** keywords must consist of a list of one or more help display names. These names must be separated by commas, and the final one must be terminated with a new line character. The value of the **Name** keyword is a single help display name, 16 characters or less. The value of the **Wlabel**, **Llabel**, **Slabel**, and **Title** keywords are strings enclosed in double quotes (" ").

The value of the **Text** keyword is text in ADF format (see *ADF*(4)). The following embedded codes are recognized:

\CEN	Center the line
\IND	Indent to the next tab stop
\UL	Begin underlining

\US\	End underlining
\BL\	Begin bold text (reverse video)
\BS\	End bold text
\EOT\	End of text

EXAMPLE

The following is the beginning of a help file, which might be used for a word processing help facility. It is recommended that all help files include a help display called "Using help," as in this example.

```
Wlabel = "Word processor help"
Contents = Using help, Getting started, Cursor,
Insert, Edit, Format
Name = Using help
LLlabel = " USING     HELP"
Slabel = "  HELP"
Branch = Using help, Getting started
Title = "How to use the HELP facility"
Text = You can use the HELP facility in two different ways:

Normal help displays consist of a description which \
displayed in a window. If the description doesn't fit \
in the window, the ROLL UP and ROLL DOWN keys may be \
used to view the rest of the display. The screen \
labeled keys at the bottom of the display contain the \
names of other help displays. Press one of these function \
keys to view a different help display.

Press function key F1 (labeled\UL\TABLE OF
CONTENTS\US\ on the screen) \
to see a listing of all available help \
displays. Select the help display you want with the \
cursor and press ENTER.

In either case, pressing EXIT ends the help display.\EOT\

Name = Getting started
Llabel = "GETTING STARTED"
Slabel = "STARTING"
Branch = Using help, Cursor, Insert, Edit, Format
Title = "Starting to use the word processor"
Text =
                .
                .
                .
```

Note that the returns are all escaped with the backslash (\), except for the hard returns at the end of paragraphs.

SEE ALSO

message(3T), ADF(4).

CAVEATS

Uahelp arbitrarily limits help files to 100 distinct displays, and each display is limited to 100 lines.

NAME

uaupd – update user agent special files

SYNOPSIS

uaupd -r ObjectName [**-a** UpdateFile] filename

DESCRIPTION

Uaupd updates the special file named in the command line. This file is assumed to reside in the directory **/usr/lib/ua**.

The **-r** option must be specified, and removes the entry associated with the given *ObjectName* from the special file.

The **-a** option adds the contents of the *UpdateFile* to the special file. The format of the user agent special files is described in *ua*(4).

SEE ALSO

ua(4).

NAME

umask – set file-creation mode mask

SYNOPSIS

umask [ooo]

DESCRIPTION

The user file-creation mode mask is set to *ooo*. The three octal digits refer to read/write/execute permissions for *owner*, *group*, and *others*, respectively (see *chmod*(2) and *umask*(2)). The value of each specified digit is subtracted from the corresponding "digit" specified by the system for the creation of a file (see *creat*(2)). For example, **umask 022** removes *group* and *others* write permission (files normally created with mode **777** become mode **755**; files created with mode **666** become mode **644**). Umask 022 is the default on the UNIX PC.

If *ooo* is omitted, the current value of the mask is printed.

Umask is recognized and executed by the shell.

SEE ALSO

chmod(1), sh(1), chmod(2), creat(2), umask(2).

NAME

umodem – remote file transfer program for CP/M terminals

SYNOPSIS

umodem – [rb | rt | sb | st] [q] [l] [m] [d] [y] [7] *filename*

DESCRIPTION

Umodem cooperates with the MODEM.COM, YAM.COM, or similar program, running on a CP/M-based intelligent terminal, to perform a file transfer. The integrity of the transfer is enhanced by use of a block checksum for error detection, and block retransmission for error correction.

Umodem requires exactly one of the following commands:

rb Receive Binary–transfer a file *from* the terminal, in raw binary mode. Every byte of the file will be transferred intact. This mode is usually used to transfer, for example, .COM files.

rt Receive Text–transfer a file *from* the terminal, in text mode. In this mode the program attempts to convert from the CP/M text file format to the UNIX format on-the-fly, by stripping carriage-return characters, and by ceasing to store data after a control-Z is detected.

sb Send Binary–transfer a file *to* the terminal, in raw binary mode. Every byte of the file will be transferred intact. This mode is usually used to transfer, for example, .COM files.

st Send Text–transfer a file *to* the terminal, in text mode. In this mode the program attempts to convert from the UNIX text file format to the CP/M format on-the-fly, by adding carriage-return characters, and by appending a control-Z to the end of the file.

In addition, *umodem* recognizes the following options:

q Quiet option–the initial "boiler plate" of program name, file size, etc., is suppressed.

l Logfile option–enables logging the progress of the file transfer. This option is primarily useful for debugging.

m "Mung-mode" option–unless this option is specified, an attempt to receive a *filename* that already exists will be denied. With this option, the existing file is overwritten.

d Delete the logfile, if it exists, before starting.

y Display file status (size) information only.

7 Seven-bit transfer option–strip off the high-order bit of each byte before it is sent (**-st** case) or stored (**-rt** case). This option is valid only for text-mode transfers.

EXAMPLES

To transfer MODEM.COM (an executable binary file) to UNIX:

umodem -rb modem.com

To transfer MYDOC.TXT (a WordStar™ text file) to UNIX, and get rid of the high–order formatting bits that WordStar™ loves to embed in the file:

umodem -rt7 mydoc.txt

To transfer **foo.c** (a UNIX C–source file) to the CP/M terminal:

umodem -st foo.c

FILES

$HOME/umodem.log created or appended to if the **–l** option is specified.

SEE ALSO

MODMPROT.001–Ward Christensen's description of the MODEM protocol

MODEM7xx.DOC–Documentation for the MODEM7 series of CP/M smart terminal programs, written in 8080 assembly language

YAMDOC.RNO–Documentation for the YAM smart terminal program, written in BDS C.

BUGS

The program supports only the checksum block error check, and not the more robust CRC.

The program supports neither the MODEM7 nor the YAM batch file transfer protocols. Only single file transfers are supported.

NAME

uname – print name of current UNIX system

SYNOPSIS

uname [**–snrvma**]

DESCRIPTION

Uname prints the current system name of UNIX on the standard output file. It is mainly useful to determine what system one is using. The options cause selected information returned by *uname*(2) to be printed:

–s print the system name (default).

–n print the nodename (the nodename may be a name that the system is known by to a communications network).

–r print the operating system release.

–v print the operating system version.

–m print the machine hardware name.

–a print all the above information.

Arguments not recognized default the command to the **–s** option.

SEE ALSO

uname(2).

NAME

unget – undo a previous get of an SCCS file

SYNOPSIS

unget [**–r**SID] [**–s**] [**–n**] files

DESCRIPTION

Unget undoes the effect of a **get –e** done prior to creating the intended new delta. If a directory is named, *unget* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Keyletter arguments apply independently to each named file.

–r*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the "new delta"). The use of this keyletter is necessary only if two or more outstanding *get*s for editing on the same SCCS file were done by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

–s Suppresses the printout, on the standard output, of the intended delta's *SID*.

–n Causes the retention of the gotten file which would normally be removed from the current directory.

SEE ALSO

delta(1), get(1), sact(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

uniq – report repeated lines in a file

SYNOPSIS

uniq [**–udc** [+n] [–n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. *Input* and *output* should always be different. Note that repeated lines must be adjacent in order to be found; see *sort*(1). If the **–u** flag is used, just the lines that are not repeated in the original file are output. The **–d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **–u** and **–d** mode outputs.

The **–c** option supersedes **–u** and **–d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

–*n* The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+*n* The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

comm(1), sort(1).

NAME

units – conversion program

SYNOPSIS

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
        * 2.540000e+00
        / 3.937008e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have: 15 lbs force/in2
You want: atm
        * 1.020689e+00
        / 9.797299e-01
```

Units only does multiplicative scale changes; thus it can convert Kelvin to Rankine, but not Celsius to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter,
c	speed of light,
e	charge on an electron,
g	acceleration of gravity,
force	same as **g**,
mole	Avogadro's number,
water	pressure head per unit height of water,
au	astronomical unit.

Pound is not recognized as a unit of mass; **lb** is. Compound names are run together, (e.g. **lightyear**). British units that differ from their U.S. counterparts are prefixed thus: **brgallon**. For a complete list of units, type:

```
cat /usr/lib/unittab
```

FILES

/usr/lib/unittab

NAME

uucp, uulog, uuname – UNIX-to-UNIX copy

SYNOPSIS

uucp [options] source-files destination-file

uulog [options]

uuname [**–l**]

DESCRIPTION

Uucp.

Uucp copies files named by the *source-file* arguments to the *destination-file* argument. A file name may be a path name on your machine, or may have the form:

system-name!path-name

where *system-name* is taken from a list of system names which *uucp* knows about. The *system-name* may also be a list of names such as

system-name!system-name!...!system-name!path-name

in which case an attempt is made to send the file via the specified route, and only to a destination in PUBDIR (see below). Care should be taken to insure that intermediate nodes in the route are willing to forward information.

The shell metacharacters **?**, ***** and **[. . .]** appearing in *path-name* will be expanded on the appropriate system.

Path names may be one of:

(1) a full path name;

(2) a path name preceded by *~user* where *user* is a login name on the specified system and is replaced by that user's login directory;

(3) a path name preceded by *~/user* where *user* is a login name on the specified system and is replaced by that user's directory under PUBDIR;

(4) anything else is prefixed by the current directory.

If the result is an erroneous path name for the remote system the copy will fail. If the *destination-file* is a directory, the last part of the *source-file* name is used.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*:

–d Make all necessary directories for the file copy (default).

–f Do not make intermediate directories for the file copy.

–c Use the source file when copying out rather than copying the file to the spool directory (default).

–C Copy the source file to the spool directory.

–m*file* Report status of the transfer in *file*. If *file* is omitted, send mail to the requester when the copy is completed.

−n*user* Notify *user* on the remote system that a file was sent.

−e*sys* Send the *uucp* command to system *sys* to be executed there. (Note: this will only be successful if the remote machine allows the *uucp* command to be executed by **/usr/lib/uucp/uuxqt**.)

Uucp returns on the standard output a string which is the job number of the request. This job number can be used by *uustat* to obtain status or terminate the job.

Uulog.

Uulog queries a summary log of *uucp* and *uux*(1C) transactions in the file **/usr/spool/uucp/LOGFILE.**

The options cause *uulog* to print logging information:

−s*sys* Print information about work involving system *sys*.

−u*user* Print information about work done for the specified *user*.

Uuname.

Uuname lists the uucp names of known systems. The **−l** option returns the local system name.

FILES

/usr/spool/uucp	spool directory
/usr/spool/uucppublic	public directory for receiving and sending (PUBDIR)
/usr/lib/uucp/*	other data and program files

SEE ALSO

mail(1), uux(1C).

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will very likely not be able to fetch files by path name; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary path names. As distributed, the remotely accessible files are those whose names begin **/usr/spool/uucppublic** (equivalent to **~nuucp** or just **~**).

BUGS

All files received by *uucp* will be owned by *uucp*.

The **−m** option will only work sending files or receiving a single file. Receiving multiple files specified by special shell characters **?** * **[. . .]** will not activate the **−m** option.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

uustat [options]

DESCRIPTION

Uustat will display the status of, or cancel, previously specified *uucp* commands, or provide general status on *uucp* connections to other systems. The following *options* are recognized:

–j*jobn* Report the status of the *uucp* request *jobn*. If **all** is used for *jobn*, the status of all *uucp* requests is reported. If *jobn* is omitted, the status of the current user's *uucp* requests is reported.

–k*jobn* Kill the *uucp* request whose job number is *jobn*. The killed *uucp* request must belong to the person issuing the *uustat* command unless one is the super-user.

–r*jobn* Rejuvenate *jobn*. That is, *jobn* is touched so that its modification time is set to the current time. This prevents *uuclean* from deleting the job until the job's modification time reaches the limit imposed by *uuclean*.

–c*hour* Remove the status entries which are older than *hour* hours. This administrative option can only be initiated by the user **uucp** or the super-user.

–u*user* Report the status of all *uucp* requests issued by *user*.

–s*sys* Report the status of all *uucp* requests which communicate with remote system *sys*.

–o*hour* Report the status of all *uucp* requests which are older than *hour* hours.

–y*hour* Report the status of all *uucp* requests which are younger than *hour* hours.

–m*mch* Report the status of accessibility of machine *mch*. If *mch* is specified as **all**, then the status of all machines known to the local *uucp* are provided.

–M*mch* This is the same as the *–m* option except that two times are printed: the time that the last status was obtained and the time that the last successful transfer to that system occurred.

–O Report the *uucp* status using the octal status codes listed below. If this option is not specified, the verbose description is printed with each *uucp* request.

–q List the number of jobs and other control files queued for each machine and the time of the oldest and youngest file queued for each machine. If a lock file exists for that system, its date of creation is listed.

When no options are given, *uustat* outputs the status of all *uucp* requests issued by the current user. Note that only one of the

options **-j**, **-m**, **-k**, **-c**, **-r**, can be used with the rest of the other options.

For example, the command:

uustat -uhdc -smhtsa -y72

will print the status of all *uucp* requests that were issued by user *hdc* to communicate with system *mhtsa* within the last 72 hours. The meanings of the job request status are:

job-number user remote-system command-time status-time status

where the *status* may be either an octal number or a verbose description. The octal code corresponds to the following description:

OCTAL	STATUS
000001	the copy failed, but the reason cannot be determined
000002	permission to access local file is denied
000004	permission to access remote file is denied
000010	bad *uucp* command is generated
000020	remote system cannot create temporary file
000040	cannot copy to remote directory
000100	cannot copy to local directory
000200	local system cannot create temporary file
000400	cannot execute *uucp*
001000	copy (partially) succeeded
002000	copy finished, job deleted
004000	job is queued
010000	job killed (incomplete)
020000	job killed (complete)

The meanings of the machine accessibility status are:

system-name time status

where *time* is the latest status time and *status* is a self-explanatory description of the machine status.

FILES

/usr/spool/uucp	spool directory
/usr/lib/uucp/L_stat	system status file
/usr/lib/uucp/R_stat	request status file

SEE ALSO

uucp(1C).

NAME

uuto, uupick – public UNIX-to-UNIX file copy

SYNOPSIS

uuto [options] source-files destination
uupick [**–s** system]

DESCRIPTION

Uuto sends *source-files* to *destination*. *Uuto* uses the *uucp*(1C) facility to send files, while it allows the local system to control the file access. A source-file name is a path name on your machine. Destination has the form:

system!*user*

where *system* is taken from a list of system names that *uucp* knows about (see *uuname*). *Logname* is the login name of someone on the specified system.

Two *options* are available:

–p Copy the source file into the spool directory before transmission.

–m Send mail to the sender when the copy is complete.

The files (or sub-trees if directories are specified) are sent to PUBDIR on *system*, where PUBDIR is a public directory defined in the *uucp* source. Specifically the files are sent to

PUBDIR/receive/*user*/*mysystem*/files.

The destined recipient is notified by *mail*(1) of the arrival of files.

Uupick accepts or rejects the files transmitted to the user. Specifically, *uupick* searches PUBDIR for files destined for the user. For each entry (file or directory) found, the following message is printed on the standard output:

from *system*: [file *file-name*] [dir *dirname*] ?

Uupick then reads a line from the standard input to determine the disposition of the file:

<new-line>	Go on to next entry.
d	Delete the entry.
m [*dir*]	Move the entry to named directory *dir* (current directory is default).
a [*dir*]	Same as **m** except moving all the files sent from *system*.
p	Print the content of the file.
q	Stop.
EOT (control-d)	Same as **q**.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

Uupick invoked with the **–s***system* option will only search the PUBDIR for files sent from *system*.

FILES

PUBDIR /usr/spool/uucppublic public directory

SEE ALSO

mail(1), uuclean(1M), uucp(1C), uustat(1C), uux(1C).

NAME

uux – UNIX-to-UNIX command execution

SYNOPSIS

uux [options] command-string

DESCRIPTION

Uux will gather zero or more files from various systems, execute a command on a specified system and then send standard output to a file on a specified system. Note that, for security reasons, many installations will limit the list of commands executable on behalf of an incoming request from *uux*. Many sites will permit little more than the receipt of mail (see *mail*(1)) via *uux*.

The *command-string* is made up of one or more arguments that look like a Shell command line, except that the command and file names may be prefixed by *system-name*!. A null *system-name* is interpreted as the local system.

File names may be one of

(1) a full path name;

(2) a path name preceded by ~*xxx* where *xxx* is a login name on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

As an example, the command

uux "!diff usg!/usr/dan/f1 pwba!/a4/dan/f1 > !f1.diff"

will get the **f1** files from the "usg" and "pwba" machines, execute a *diff* command and put the results in **f1.diff** in the local directory.

Any special shell characters such as <>;| should be quoted either by quoting the entire *command-string*, or quoting the special characters as individual arguments.

Uux will attempt to get all files to the execution system. For files which are output files, the file name must be escaped using parentheses. For example, the command

uux a!uucp b!/usr/file \(c!/usr/file\)

will send a *uucp* command to system "a" to get **/usr/file** from system "b" and send it to system "c".

Uux will notify you if the requested command on the remote system was disallowed. The response comes by remote mail from the remote machine.

The following *options* are interpreted by *uux*:

– The standard input to *uux* is made the standard input to the *command-string*.

–n Send no notification to user.

–m*file* Report status of the transfer in *file*. If *file* is omitted, send mail to the requester when the copy is completed.

Uux returns an ASCII string on the standard output which is the job number. This job number can be used by *uustat* to obtain the status or terminate a job.

FILES

/usr/lib/uucp/spool	spool directory
/usr/lib/uucp/*	other data and programs

SEE ALSO

uuclean(1M), uucp(1C).

BUGS

Only the first command of a shell pipeline may have a *systemname*!. All other commands are executed on the system of the first command.

The use of the shell metacharacter * will probably not do what you want it to do. The shell tokens << and >> are not implemented.

NAME

val – validate SCCS file

SYNOPSIS

val –
val [–**s**] [–**r**SID] [–**m**name] [–**y**type] files

DESCRIPTION

Val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of keyletter arguments, which begin with a –, and named files.

Val has a special argument, –, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

Val generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

The keyletter arguments are defined as follows. The effects of any keyletter argument apply independently to each named file on the command line.

–s The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

–r*SID* The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., **r1** is ambiguous because it physically does not exist but implies 1.1, 1.2, etc. which may exist) or invalid (e. g., **r**1.0 or **r**1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

–m*name* The argument value *name* is compared with the SCCS %M% keyword in *file*.

–y*type* The argument value *type* is compared with the SCCS %Y% keyword in *file*.

The 8-bit code returned by *val* is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;
bit 2 = corrupted SCCS file;
bit 3 = can't open file or file not SCCS;
bit 4 = *SID* is invalid or ambiguous;

bit 5 = *SID* does not exist;
bit 6 = %Y%, **-y** mismatch;
bit 7 = %M%, **-m** mismatch;

Note that *val* can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned—a logical **OR** of the codes generated for each command line and file processed.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

Val can process up to 50 files on a single command line. Any number above 50 will produce a **core** dump.

NAME

vc – version control

SYNOPSIS

vc [**–a**] [**–t**] [**–c**char] [**–s**] [keyword=value ... keyword=value]

DESCRIPTION

The *vc* command copies lines from the standard input to the standard output under control of its *arguments* and *control statements* encountered in the standard input. In the process of performing the copy operation, user declared *keywords* may be replaced by their string *value* when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests (in control statements) of keyword values specified in control statements or as *vc* command arguments.

A control statement is a single line beginning with a control character, except as modified by the **–t** keyletter (see below). The default control character is colon (**:**), except as modified by the **–c** keyletter (see below). Input lines beginning with a backslash (\) followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or less alphanumerics; the first must be alphabetic. A value is any ASCII string that can be created with *ed*(1); a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Replacement of keywords by values is done whenever a keyword surrounded by control characters is encountered on a version control statement. The **–a** keyletter (see below) forces replacement of keywords in *all* lines of text. An uninterpreted control character may be included in a value by preceding it with \. If a literal \ is desired, then it too must be preceded by \.

Keyletter Arguments

–a Forces replacement of keywords surrounded by control characters with their assigned value in *all* text lines and not just in *vc* statements.

–t All characters from the beginning of a line up to and including the first *tab* character are ignored for the purpose of detecting a control statement. If one is found, all characters up to and including the *tab* are discarded.

–c*char* Specifies a control character to be used in place of **:**.

–s Silences warning messages (not error) that are normally printed on the diagnostic output.

Version Control Statements

:dcl keyword[, ..., keyword]

Used to declare keywords. All keywords must be declared.

:asg keyword=value

Used to assign values to keywords. An **asg** statement overrides the assignment for the corresponding keyword on the *vc* command line and all previous **asg**'s for that keyword. Keywords declared, but not assigned values have null values.

:if condition

:

:

:end

Used to skip lines of the standard input. If the condition is true all lines between the *if* statement and the matching *end* statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control statements. Note that intervening *if* statements and matching *end* statements are recognized solely for the purpose of maintaining the proper *if-end* matching.

The syntax of a condition is:

```
<cond>   ::= [ "not" ] <or>
<or>     ::= <and> | <and> "|" <or>
<and>    ::= <exp> | <exp> "&" <and>
<exp>    ::= "(" <or> ")" | <value> <op>
<value>
<op>     ::= "=" | "!=" | "<" | ">"
<value>  ::= <arbitrary ASCII string> | <numeric
string>
```

The available operators and their meanings are:

=	equal
!=	not equal
&	and
|	or
>	greater than
<	less than
( )	used for logical groupings
not	may only occur immediately after the *if*, and when present, inverts the value of the entire condition

The > and < operate only on unsigned integer values (e. g.: 012 > 12 is false). All other operators take strings as arguments (e. g.: 012 != 12 is true). The precedence of the operators (from highest to lowest) is:

```
= != > <      all of equal precedence
&
|
```

Parentheses may be used to alter the order of precedence. Values must be separated from operators or parentheses by at least one blank or tab.

::text
Used for keyword replacement on lines that are copied to the standard output. The two leading control characters are removed, and keywords surrounded by control characters in text are replaced by their value before the line is copied to the output file. This action is independent of the **-a** keyletter.

:on

:off
Turn on or off keyword replacement on all lines.

:ctl char
Change the control character to char.

:msg message
Prints the given message on the diagnostic output.

:err message
Prints the given message followed by:

ERROR: err statement on line ... (915)

on the diagnostic output. *Vc* halts execution, and returns an exit code of 1.

DIAGNOSTICS

Use *help*(1) for explanations.

EXIT CODES

0 – normal
1 – any error

NAME

vi, view – screen oriented (visual) display editor based on ex

SYNOPSIS

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

DESCRIPTION

Vi (visual) is a display oriented text editor based on *ex*(1) *View* is synonymous with *vi*. *Ex* and *vi* run the same code; it is possible to get to the command mode of *ex* from within *vi* and vice-versa.

Note that the ability to edit encrypted files is present only in the domestic (U.S.) version of the UNIX PC software.

COMMANDS

The following summarizes the *vi* commands and procedures. The *Introduction to Display Editing with Vi* provides full details on using *vi*.

NOTATION AND SPECIAL KEYS

^	Denotes the CONTROL key (Ctrl on the UNIX PC) to be held down while the following character is typed.
↑	Used to show the caret (^) should be typed.
[n]	Optional number of repetitions preceding a command. Do not type [ ]. In most cases omitting *n* defaults to one.
object	The text object—(character, word, sentence, paragraph, or line) that a command operates on.
:	A prefix to a set of commands for file and option manipulation and escapes to the shell. The **:** and later keystrokes appear at the bottom of the screen. The command is terminated with a <**CR**> or <**ESC**>.
<**ESC**>	ESCAPE key (Esc on the UNIX PC) used to return to command mode. Type <**ESC**> when you are not sure of the current mode. Causes a beep if already in command mode (harmless).
<**CR**>	Carriage RETURN key.
BS	BACKSPACE key. ^**H** on terminals without a backspace key.
DELETE	Sometimes labeled **DEL**, **BREAK**, or **RUBOUT** (shift of the Esc key on the UNIX PC). This key generates an interrupt that tells the editor to stop what it is doing.

ENTERING THE VI EDITOR

Note: Follow entry with <CR>.

vi *file*	Edit at first line of *file*
vi	Edit a new empty *file*
vi + *n file*	Edit at *n* line in *file*
vi + *file*	Edit at last line in *file*
vi −r	List saved files
vi −r *file*	Recover *file* and edit saved *file*
vi *file1, file2, ...*	Edit *file1*; *file2*; ... (after editing *file1* enter **:n** for each remaining file)
vi −t *tag*	Edit at *tag* file in *tags* file
vi +/*pat file*	Search for and edit at *pattern* in *file*
view *file*	Read only view of file

LEAVING VI EDITOR

:q<CR>	Quit *vi* when no changes have occurred since last write
:q!<CR>	Quit *vi*, do not save changes since last write
:wq<CR>	Write and quit (exit *vi*, saving changes)
ZZ	Write and quit (exit *vi*, saving changes)

POSITIONING THE CURSOR

File Positioning

/n/^F	Forward */n/* full screens
/n/^B	Backward screens
/n/^D	Scroll down (default is half screen)
/n/^U	Scroll up (default is half screen)
/n/^E	Scroll down 1 line
/n/^Y	Scroll up 1 line
/n/G	Go to line *n* (default is last line of file)
/n//pat	Go to next line matching *pat*
/n/?pat	Previous line matching *pat*
/n/n	Repeat last / or ?
/n/N	Reverse last / or ?
/n//pat/+m	*m*th line after *pat*
/n/?pat?−m	*m*th line before *pat*

Screen Positioning

/n/H	To *n*th line from top of display. Without *n*, to top
/n/L	To *n*th line from bottom of display. Without *n*, to bottom
M	To middle line of display

Line Positioning

0	Beginning of line

[n]$	End of line
[n]+	Next line, at first non-white
[n]–	Previous line, at first non-white
[n]<CR>	Return, same as +
[n]↓| or **j**	Next line, same column
[n]↑| or **k**	Previous line, same column

Character Positioning Within a Line

[n]↑	First non-white
*[n]***h** or →	Forward one character
*[n]***l** or ←	Backward one character
*[n]*spacebar	Same as →
*[n]*backspace	Backwards one character
[n]^**H**	Same as ← or backspace
*[n]***f***x*	Find *x* forward
*[n]***F***x*	Find *x* backward
*[n]***t***x*	Move up to *x* forward
*[n]***T***x*	Move up to *x* backward
*[n]***;**	Repeat last **f**, **F**, **t**, or **T**
*[n]***,**	Inverse of **;**
[n]|	Move to specified column number *n*

Word Positioning

*[n]***w**	Move forward to beginning of word. Punctuation and strings of punctuation count as words.
*[n]***b**	Move back to beginning of word. Punctuation and strings of punctuation count as words.
*[n]***e**	Move forward to end of word. Punctuation and strings of punctuation count as words.
*[n]***W**	Move forward to beginning of word. Punctuation ignored.
*[n]***B**	Move back to beginning of word. Punctuation ignored.
*[n]***E**	Move forward to end of word. Punctuation ignored.

Sentence, Paragraph, Heading Positioning

[n])	Forward to next sentence
[n](	Back a sentence
[n]}	Forward to next paragraph
[n]{	Back a paragraph

[n]]]	Forward to next heading
[n] [[	Back a heading

CREATING TEXT

a*text*<ESC>	Append after cursor, until <ESC>
i*text*<ESC>	Insert before cursor
A*text*<ESC>	Append at end of line
I*text*<ESC>	Insert before first non-blank
o*text*<ESC>	Open line below
O*text*<ESC>	Open above

MAKING CORRECTIONS DURING TEXT CREATION

^W	Erase last word during an insert
kill	Kill the insert on this line (usually @, **^X**, or **^U**)
*[n]*BS	Erase last character
^H	Erase last character
****	Escapes **^H**, your erase and kill
<ESC>	Ends insertion, back to command mode
^?	Interrupt, terminates insert
^D	Backtab over *autoindent*
↑^D	Kill *autoindent*, save for next
0^D	... but at margin next also
^V	Quote non-printing character

MODIFYING TEXT

Changing Text

~	Switch character from lowercase to uppercase and vice versa
*[n]***C***text*<ESC>	Change from cursor to end of line (same as **c$**)
*[n]***R***text*<ESC>	Replace characters
*[n]***S***text*<ESC>	Substitute on lines
*[n]*c*obj**text***<ESC>	Change the specified object (word) to the following *text*
*[n]***r***x*	Replace character with *x*
*[n]***s***text*<ESC>	Replace a character with a *text* string
*[n]***cc***text*<ESC>	Change a whole line

Deleting Text

D	Delete from cursor to end of line
*[n]***x**	Delete a character
*[n]***X**	Delete character to left of cursor
*[n]***d***(object)*	Delete the specified *object* (word, sentence, paragraph, etc.)

*[n]*dd	Delete a line

Moving Text

"*r*	Named register *r* that save delete commands. Legal values of *r* are letters **a** through **z**.
"*r*p	Puts deleted text from registers "*r* after or below cursor
"*r*P	Puts deleted text from registers "*r* before or above cursor
p	Puts last deleted text after or below cursor
P	Puts last deleted text before or above cursor

Copying Text

"*r*	Named register *r* that can precede a yank command. Legal values of *r* are letters **a** through **z**.
y*[n]*object	Yanks a copy of the following object into a register
*[n]*Y	Yanks a copy of the current line into a register
*[n]*yy	Same as Y
"*r*p	Puts yanked text from register "*r* after or below cursor
"*r*P	Puts yanked text from register "*r* before or above cursor
p	Puts last yanked text after or below cursor
P	Puts last yanked text before or above cursor

UNDOING, REDOING, RETRIEVING

u	Undo last change
U	Restore current line
.	Repeat last change
"*h*p	Retrieve one of last 9 deletes; *h* is a hidden register numbered 1 through 9. Retrieved in reverse order.

DOING GLOBAL SEARCHES AND CHANGES

Note: Follow entry with <CR>.

:g/*text*	Move cursor to last line in file with *text*
:g/*text*/p	Print all lines with *text*
:g/*text*/nu	Print all lines and line numbers with *text*
:*[m]*,*[n]*g/*text*	Move cursor to *n* line in file with *text*
:*[m]*,*[n]*g/*text*/p	Print all lines with *text* from line *m* to *n*
:*[m]*,*[n]*g/*text*/nu	Print all lines and line numbers with *text* from line *m* to *n*
:g/*text*/s//*newtext*	Change first appearance of *text* in each line in

file to *newtext*

:g/*text***/s//***newtext***/p**
Change first appearance of *text* in each line in file to *newtext* and print each changed line

:g/*text***/s//***newtext***/c**
List one at a time each line with *text* and change as required to *newtext* using a **y**<CR>

:*[m]***,***[n]***g/***text***/s//***newtext***
Change first appearance of *text* in each line in file to *newtext*

:*[m]***,***[n]***g/***text***/s//***newtext***/p**
Change first appearance of *text* in lines from *m* to *n* to *newtext* and print each changed line

:*[m]***,***[n]***g/***text***/s//***newtext***/c**
List one at a time each line with *text* from *m* to *n* and change as required to *newtext* using a **y**<CR>

MANIPULATING FILES

Copy From Another File

:r *file*<CR> Copy *file* into buffer after current line

:*[n]***r** *file*<CR> Copy *file* to buffer after *n*th line

Copy To Another File

Note: Follow entry with <CR>.

:w *file* Write the current file to *file*

:w! *file* Overwrite existing *file* with *file*

:w>>*file* Add current file to end of *file*

:*[m]***,***[n]***w** *file* Write lines *m* through *n* to *file*

:*[m]***,***[n]***w!** *file* Overwrite existing *file* with *file* containing lines *m* through *n*

:*[m]***,***[n]***w>>***file* Add lines *m* through *n* to end of *file*

Edit Current File

:w<CR> Write changes to current file

:w *file*<CR> Write *file* to current unnamed file

"e!<CR> Reedit current file, discarding changes since last write

:f<CR> Show current file and line

^G Synonym for **:f**

:ta *tag*<CR> To tag file entry *tag*

^] **:ta**, following word is *tag*

Edit Other Files From Current File

:e *file*<CR> Edit *file* when write has occurred in current file, return to shell after edit, changes not lost in current file

:e! *file*<CR>	Edit *file* when no write has occurred in current file, return to shell after edit, changes list in current file
:e + *name*<CR>	Edit starting at end
:e + *n*<CR>	Edit starting at line *n*
:n<CR>	Edit next file in list when *vi* was called with more than one file
:n *args*<CR>	Specify new list of files to be edited
:e #<CR>	Edit alternate file when two files are being edited
^↑	Synonym for **:e #**.

ESCAPING TO THE SHELL

:sh<CR>	Start a separate shell (to run several commands), return with ^D
:!*command*<CR>	Run one shell *command*, then return to current buffer

MARKING AND RETURNING

``	Previous context
''	... at first non-white in line
m*x*	Mark position with letter *x*
`*x*	to mark *x*
'*x*	... at first non-white in line

MISCELLANEOUS OPERATIONS

.	Repeat the last append, insert, open, delete, change, or put command
~	Switch character from lowercase to uppercase and vice versa
^?	Delete or rubout interrupts
i<CR><ESC>	Split a line before the cursor
a<CR><ESC>	Split a line after the cursor
^L	Reprint screen if **^?** scrambles it
J	Join lines
:nu<CR>	Line number cursor is on
xp	Switch characters

SETTING OPTIONS

Initializing Options

:set *x*<CR>	Enable option *x*
:set no*x*<CR>	Disable option *x*
:set *x*=*val*<CR>	Assign a value to *x* option
:set<CR>	Show changed options
:set all<CR>	Show all options

:set *x*?<CR> Show value of option *x*

Options

autoindent, ai (default: noai)
When on, in the append, change, insert, open, or substitute mode a new line will be started at same indent as previous line.

audoprint, ap (default: ap)
Causes the current line to be printed after each delete, copy, join, move, substitute, t, undo or shift command. This has the same effect as supplying a trailing **p** to each such command. The *autoprint* is suppressed in globals and only applies to the last of many commands on a line.

autowrite, aw (default: noaw)
Causes the contents of the buffer to be written to the current file (if you have modified it) and gives a next, rewind, tab, or ! command, or a ^↑ (switch files) or ^] (tag goto) command. Note: the command does not autowrite. In each case, there is an equivalent way of switching when the *autowrite* option is set to avoid the autowrite (ex for next, rewind! for rewind, tag! for tag, shell for !, and :e #nd for a :ta! command).

beautify, bf (default: nobeautify)
Causes all control characters except tab, newline, and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. The *beautify* option does not apply to command input.

directory, dir (default: dir=/tmp)
Specifies the directory in which *vi* places its buffer file. If this directory is not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible (default: noedcompatible)
Causes the presence or absence of **g** and **c** suffixes on substitute commands to be remembered and to be toggled by repeating the suffixes. The suffix **r** makes the substitution be as in the ~ command, instead of line &.

errorbells, eb (default: noeb)
Error messages are preceded by a bell. Bell ringing in *open* and *visual* mode on errors is not suppressed by setting *noeb*. If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht (default: ht=8)
Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic (default: noic)
All uppercase characters in the text are mapped to lower case in regular expression matching and vice versa, except in character class specifications.

lisp (default: nolisp)
The *autoindent* option indents appropriately for *lisp code, and the* ( ), { }, [[, and]] commands in *open* and *visual* modes are modified to have meaning for *lisp*.

list (default: nolist)
All printed lines will be displayed more unambiguously, showing tabs and end-of-lines as in the *list* command.

magic (default: magic for *vi*)
If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only ↑ and $ having special effects. In addition, the metacharacters ~ and & of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a \.

mesg (default: mesg)
Causes write permission to be turned off to the terminal while you are in *visual* mode if *nomesg* is set.

number, nu (default: nonumber)
Causes all output lines to be printed with line numbers. In addition, each input line will be prompted for by supplying the line number it will have.

open (default: open)
If *noopen*, the commands *open* and *visual* are not permitted.

optimize opt (default: optimize)
Throughput of text is expedited by setting the terminal not to do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para (default: para=IPLPPPQPPLIbp)
Specifies the paragraphs for the { and } operations in *open* and *visual* mode. The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt (default: prompt)
Command mode input is prompted for with a colon (:).

readonly (default: noreadonly)
Set by *chmod* shell command to allow read but no write.

redraw (default: noredraw)
The editor simulates (using great amounts of output) an intelligent terminal on a dumb terminal (e.g., during insertions in *visual* mode the characters to the right of the cursor position are refreshed as each input character is typed). This option is useful only at very high speed.

remap (default: remap)
If on, macros are repeatedly tried until they are unchanged. For example, if **o** is mapped to **O**, and **O** is

mapped to **I**, then if *remap* is set, **o** will map to *I*; but of *noremap* is set, if will map to **O**.

report (default: report=5)
Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as global, *open*, undo, and *visual*, which have potentially more far-reaching scope, the net change in the number of lines in the buffer is presented at the end of the command subject to this same threshold. Thus, notification is suppressed during a global command on the individual commands performed.

scroll (default: scroll=½ window)
Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in *command* mode and the number of lines printed by a *command* mode **z** command (double the value of *scroll*).

sections (default: sections=SHNHH HU)
Specifies the section macros for the [[and]] operations in *open* and *visual* modes. The pairs of characters in the option's value are the names of the macros which start paragraphs.

shell, sh (default: sh=/bin/sh)
Gives the pathname of the shell forked for the shell escape command **!**, and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw (default: sw=8)
Gives the width a software tabstop used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.

showmatch, sm (default: nosm)
In *open* and *visual* modes, when a) or } is typed, the cursor moves to the matching (or { for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow (default: terminal dependent)
Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent.

tabstop, ts (default: ts=8)
The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl (default: tl=0)
Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags (default: tags=tags /usr/lib/tags)
A path of files to be used as tag files for the *tag* command.

A requested tag is searched for in the specified files, sequentially. By default, files called *tags* are searched for in the current directory and in */usr/lib* (a master file for the entire system).

term (default from environment $TERM)
The terminal type of the output device.

terse (default: noterse)
Shorter error diagnostics are produced for the experienced user.

ttytype==
Terminal type defined to system for visual mode. Can be defined before entering visual editor by TERM=type.

warn (default: warn)
Warns if there has been "[No write since last change]" before a **!** command escape.

window (default: window=speed dependent)
The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600
These are not true options but set *window* only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws (default: ws)
Searches that use regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm (default: wm=0)
Defines a margin for automatic wrapover of text during input in *open* and *visual* modes.

writeany, wa (default: nowa)
Inhibit checks normally made before write commands, allowing a write to any file which the system protection mechanism will allow.

FILES

See *ex*(1).

SEE ALSO

ex(1), edit (1), "An Introduction to Display Editing with Vi".

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

won't be broken.

Insert/delete 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 named buffers is somewhat inefficient.

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 reenter the screen editor with *vi* or *open*.

Moving the cursor backward a screen at a time does not work correctly.

The *[n]* precursor does not work for these commands: **B**, **U**, */pat*, ?*pat*, */pat*, */pat/+m*, ?*pat*?*-m*.

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait until all processes started with & have completed, and report on abnormal terminations.

Because the *wait*(2) system call must be executed in the parent process, the shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(1).

BUGS

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus can't be waited for.

NAME

wc – word count

SYNOPSIS

wc [**–lwc**] [names]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no *names* appear. It also keeps a total count for all named files. A word is a maximal string of characters delimited by spaces, tabs, or new-lines.

The options **l**, **w**, and **c** may be used in any combination to specify that a subset of lines, words, and characters are to be reported. The default is **–lwc**.

When *names* are specified on the command line, they will be printed along with the counts.

NAME

what – identify SCCS files

SYNOPSIS

what files

DESCRIPTION

What searches the given files for all occurrences of the pattern that *get*(1) substitutes for %Z% (this is @(#) at this printing) and prints out what follows until the first ", >, new-line, \, or null character. For example, if the C program in file **f.c** contains

```
char ident[] = " @(#)identification information ";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command

```
what f.c f.o a.out
```

will print

```
f.c:
        identification information
f.o:
        identification information
a.out:
        identification information
```

What is intended to be used in conjunction with the command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually.

SEE ALSO

get(1), help(1).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

It's possible that an unintended occurrence of the pattern @(#) could be found just by chance, but this causes no harm in nearly all cases.

NAME

who – who is on the system

SYNOPSIS

who [**-uTlpdbrtas**] [file]

who am i

DESCRIPTION

Who can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process-ID of the command interpreter (shell) for each current UNIX user. It examines the **/etc/utmp** file to obtain its information. If *file* is given, that file is examined. Usually, *file* will be **/etc/wtmp**, which contains a history of all the logins since the file was last created.

Who with the **am i** option identifies the invoking user.

Except for the default **-s** option, the general format for output entries is:

name [state] line time activity pid [comment] [exit]

With options, *who* can list logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the *init* process. These options are:

-u This option lists only those users who are currently logged in. The *name* is the user's login name. The *line* is the name of the line as found in the directory **/dev**. The *time* is the time that the user logged in. The *activity* is the number of hours and minutes since activity last occurred on that particular line. A dot (**.**) indicates that the terminal has seen activity in the last minute and is therefore "current". If more than twenty-four hours have elapsed or the line has not been used since boot time, the entry is marked old. This field is useful when trying to determine whether a person is working at the terminal or not. The *pid* is the process-ID of the user's shell. The *comment* is the comment field associated with this line as found in **/etc/inittab** (see *inittab*(4)). This can contain information about where the terminal is located, the telephone number of the dataset, type of terminal if hard-wired, etc.

-T This option is the same as the **-u** option, except that the *state* of the terminal line is printed. The *state* describes whether someone else can write to that terminal. A + appears if the terminal is writable by anyone; a - appears if it is not. **Root** can write to all lines having a + or a - in the *state* field. If a bad line is encountered, a **?** is printed.

-l This option lists only those lines on which the system is waiting for someone to login. The *name* field is **LOGIN** in such cases. Other fields are the same as for user entries except that the *state* field doesn't exist.

-p This option lists any other process which is currently active and has been previously spawned by *init*. The *name* field

is the name of the program executed by *init* as found in **/etc/inittab**. The *state*, *line*, and *activity* fields have no meaning. The *comment* field shows the *id* field of the line from **/etc/inittab** that spawned this process. See *inittab*(4).

-d This option displays all processes that have expired and not been respawned by *init*. The *exit* field appears for dead processes and contains the termination and exit values (as returned by *wait*(2)), of the dead process. This can be useful in determining why a process terminated.

-b This option indicates the time and date of the last reboot.

-r This option indicates the current *run-level* of the *init* process.

-t This option indicates the last change to the system clock (via the *date*(1) command) by **root**. See *su*(1).

-a This option processes **/etc/utmp** or the named *file* with all options turned on.

-s This option is the default and lists only the *name*, *line* and *time* fields.

FILES

/etc/utmp
/etc/wtmp
/etc/inittab

SEE ALSO

init(1M) in the *UNIX System Administrator's Manual*.
date(1), login(1M), mesg(1), su(1), wait(2), inittab(4), utmp(4).

NAME

write – write to another user

SYNOPSIS

write user [line]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message:

Message from *yourname* **(tty**??**)** [*date*]**...**

to the person you want to talk to. When it has successfully completed the connection it also sends two bells to your own terminal to indicate that what you are typing is being sent.

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal or an interrupt is sent. At that point *write* writes **EOT** on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *line* argument may be used to indicate which line or terminal to send to (e.g., **tty00**); otherwise, the first instance of the user found in **/etc/utmp** is assumed and the following message posted:

user is logged on more than one place.
You are connected to *"terminal"*.
Other locations are:
terminal

Permission to write may be denied or granted by use of the *mesg*(1) command. Writing to others is normally allowed by default. Certain commands, in particular *nroff*(1) and *pr*(1) disallow messages in order to prevent interference with their output. However, if the user has super-user permissions, messages can be forced onto a write inhibited terminal.

If the character **!** is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first *write* to another user, wait for them to *write* back before starting to send. Each person should end a message with a distinctive signal (i.e., **(o)** for "over") so that the other person knows when to reply. The signal **(oo)** (for "over and out") is suggested when conversation is to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute **!**

SEE ALSO

mail(1), mesg(1), nroff(1), pr(1), sh(1), who(1).

DIAGNOSTICS

"*user not logged in*" if the person you are trying to *write* to is not logged in.

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [flags] [command [initial-arguments]]

DESCRIPTION

Xargs combines the fixed *initial-arguments* with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the flags specified.

Command, which may be a shell file, is searched for, using one's **$PATH**. If *command* is omitted, **/bin/echo** is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or new-lines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted: Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings a backslash (\) will escape the next character.

Each argument list is constructed starting with the *initial-arguments*, followed by some number of arguments read from standard input (Exception: see **–i** flag). Flags **–i**, **–l**, and **–n** determine how arguments are selected for each command invocation. When none of these flags are coded, the *initial-arguments* are followed by arguments read continuously from standard input until an internal buffer is full, and then *command* is executed with the accumulated args. This process is repeated until there are no more args. When there are flag conflicts (e.g., **–l** vs. **–n**), the last flag has precedence. *Flag* values are:

–l*number* *Command* is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first new-line *unless* the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If *number* is omitted 1 is assumed. Option **–x** is forced.

–i*replstr* Insert mode: *command* is executed for each line from standard input, taking the entire line as a single arg, inserting it in *initial-arguments* for each occurrence of *replstr*. A maximum of 5 arguments in *initial-arguments* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters, and option **–x** is also forced. { } is assumed for *replstr* if not specified.

−n*number* Execute *command* using as many standard input arguments as possible, up to *number* arguments maximum. Fewer arguments will be used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* arguments remaining. If option **−x** is also coded, each *number* arguments must fit in the *size* limitation, else *xargs* terminates execution.

−t Trace mode: the *command* and each constructed argument list are echoed to file descriptor 2 just prior to their execution.

−p Prompt mode: the user is asked whether to execute *command* each invocation. Trace mode (**−t**) is turned on to print the command instance to be executed, followed by a **?...** prompt. A reply of **y** (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of *command*.

−x Causes *xargs* to terminate if any argument list would be greater than *size* characters; **−x** is forced by the options **−i** and **−l**. When neither of the options **−i**, **−l**, or **−n** are coded, the total length of all arguments must be within the *size* limit.

−s*size* The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If **−s** is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

−e*eofstr* *Eofstr* is taken as the logical end-of-file string. Underbar (_) is assumed for the logical **EOF** string if **−e** is not coded. **−e** with no *eofstr* coded turns off the logical **EOF** string capability (underbar is taken literally). *Xargs* reads standard input until either end-of-file or the logical **EOF** string is encountered.

Xargs will terminate if it receives a return code of **−1** from, or if it cannot execute, *command*. When *command* is a shell program, it should explicitly *exit* (see *sh*(1)) with an appropriate value to avoid accidentally returning with **−1**.

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs -i -t mv $1/{} $2/{}
```

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the end of file *log*:

```
(logname; date; echo $0 $*) | xargs >>log
```

The user is asked which files in the current directory are to be archived and archives them into *arch* (1.) one at a time, or (2.) many at a time.

```
1.  ls | xargs -p -l ar r arch
2.  ls | xargs -p -l | xargs ar r arch
```

The following will execute *diff*(1) with successive pairs of arguments originally typed as shell arguments:

```
echo $* | xargs -n2 diff
```

DIAGNOSTICS

Self explanatory.

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–vdlt**] grammar

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, **y.tab.c**, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *lex*(1) is useful for creating lexical analyzers usable by *yacc*.

If the **–v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

If the **–l** flag is given, the code produced in **y.tab.c** will *not* contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

Runtime debugging code is always generated in **y.tab.c** under conditional compilation control. By default, this code is not included when **y.tab.c** is compiled. However, when *yacc*'s **–t** option is used, this debugging code will be compiled by default. Independent of whether the **–t** option was used, the runtime debugging code is under the control of **YYDEBUG**, a preprocessor symbol. If **YYDEBUG** has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	
yacc.debug, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).

YACC– Yet Another Compiler Compiler in the *UNIX System Support Tools Guide*.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

BUGS

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always –1; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

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

No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

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. 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 which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than 5,120 bytes is presented to a member of the *exec* family.

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 *a.out*(4)).

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

10 ECHILD No child processes
A *wait*, was executed by a process that had no existing or unwaited-for child processes.

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

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

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 use an argument of a system call.

15 ENOTBLK Block device required
A non-block 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). It will also occur if an attempt is made to enable accounting when it is already enabled.

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 prefix or as an argument to *chdir*(2).

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

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*, or *kill*; reading or writing a file for which *lseek* has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

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
No process may have more than 80 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which 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 file size (2,147,483,647 bytes) or ULIMIT; see *ulimit*(2).

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.

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 the maximum number of links (1000) 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 not representable within machine precision.

35 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).

36 EIDRM Identifier Removed
This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see *msgctl*(2), *semctl*(2), and *shmctl*(2)).

DEFINITIONS

Process ID
Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID
A new process is created by a currently active process; see *fork*(2). The parent process ID of a process is the process ID of its creator.

Process Group ID
Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(2).

Tty Group ID
Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(2) and *signal*(2).

Real User ID and Real Group ID
Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID
An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec*(2).

Super-user
A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes
The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Name.

Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell. See *sh*(1). Although permitted, it is advisable to avoid the use of unprintable characters in file names.

Path Name and Path Prefix

A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

More precisely, a path name is a null-terminated character string constructed as follows:

<path-name>::=<file-name>| <path-prefix><file-name>|/
<path-prefix>::=<rtprefix>| /<rtprefix>
<rtprefix>::=<dirname>/| <rtprefix><dirname>/

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory.

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory.

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process's root directory need not be the root directory of the root file system, and is determined by the *userid* entry in */etc/passwd*. The working directory for each process is determined either by cd(1) or chdir(2).

File Access Permissions.

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;  /* operation permission struct */
ushort   msg_qnum;           /* number of msgs on q */
ushort   msg_qbytes;         /* max number of bytes on q */
ushort   msg_lspid;          /* pid of last msgsnd operation */
ushort   msg_lrpid;          /* pid of last msgrcv operation */
time_t   msg_stime;          /* last msgsnd time */
time_t   msg_rtime;          /* last msgrcv time */
time_t   msg_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Msg_perm is a ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort   cuid;    /* creator user id */
ushort   cgid;    /* creator group id */
ushort   uid;     /* user id */
ushort   gid;     /* group id */
ushort   mode;    /* r/w permission */
```

Msg_qnum is the number of messages currently on the queue. **Msg_qbytes** is the maximum number of bytes allowed on the queue. **Msg_lspid** is the process id of the last process that performed a *msgsnd* operation. **Msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **Msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions.

In the *msgop*(2) and *msgctl*(2) system call descriptions, the

permission required for an operation is interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a msqid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **msg_perm.[c]uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID matches **msg_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID does not match **msg_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;  /* operation permission struct */
ushort   sem_nsems;          /* number of sems in set */
time_t   sem_otime;          /* last operation time */
time_t   sem_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Sem_perm is a ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   cuid;    /* creator user id */
ushort   cgid;    /* creator group id */
ushort   uid;     /* user id */
ushort   gid;     /* group id */
ushort   mode;    /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1. **Sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort   semval;     /* semaphore value */
short    sempid;     /* pid of last operation */
ushort   semncnt;    /* # awaiting semval > cval */
ushort   semzcnt;    /* # awaiting semval = 0 */
```

Semval is a non-negative integer. **Sempid** is equal to the process ID of the last process that performed a semaphore operation on this semaphore. **Semncnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value. **Semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions.

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, Alter by group
00006	Read, Alter by others

Read and Alter permissions on a semid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **sem_perm.[c]uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid** and the process's effective group ID matches **sem_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid** and the process's effective group ID does not match **sem_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct   ipc_perm shm_perm;  /* operation permission struct */
int      shm_segsz;          /* size of segment */
ushort   shm_cpid;           /* creator pid */
ushort   shm_lpid;           /* pid of last operation */
short    shm_nattch;         /* number of current attaches */
time_t   shm_atime;          /* last attach time */
```

```
time_t   shm_dtime;          /* last detach time */
time_t   shm_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shm_perm is a ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort   cuid;       /* creator user id */
ushort   cgid;       /* creator group id */
ushort   uid;        /* user id */
ushort   gid;        /* group id */
ushort   mode;       /* r/w permission */
```

Shm_segsz specifies the size of the shared memory segment. **Shm_cpid** is the process id of the process that created the shared memory identifier. **Shm_lpid** is the process id of the last process that performed a *shmop*(2) operation. **Shm_nattch** is the number of processes that currently have this segment attached. **Shm_atime** is the time of the last *shmat* operation, **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions.

In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a shmid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The process's effective user ID does not match **shm_perm.[c]uid** and the process's effective group ID matches **shm_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The process's effective user ID does not match **shm_perm.[c]uid** and the process's effective group ID does not match **shm_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

SEE ALSO

intro(3).

NAME

access – determine accessibility of a file

SYNOPSIS

```
int access (path, amode)
char *path;
int amode;
```

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

Read, write, or execute (search) permission is requested for a null path name. [ENOENT]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), stat(2).

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(2). The effective user ID of the calling process must be super-user to use this call.

Path points to a path name naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

Acct will fail if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

An attempt is being made to enable accounting when it is already enabled. [EBUSY]

A component of the path prefix is not a directory. [ENOTDIR]

One or more components of the accounting file's path name do not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The file named by *path* is not an ordinary file. [EACCES]

Mode permission is denied for the named accounting file. [EACCES]

The named file is a directory. [EISDIR]

The named file resides on a read-only file system. [EROFS]

Path points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

alarm – set a process's alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

Alarm instructs the calling process's alarm clock to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2).

Alarm requests are not stacked; successive calls reset the calling process's alarm clock.

If *sec* is 0, any previously made alarm request is canceled.

RETURN VALUE

Alarm returns the amount of time previously remaining in the calling process's alarm clock.

SEE ALSO

pause(2), signal(2).

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases.

Brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased. *Sbrk* clears only the page actually allocated, starting at a page boundary.

Brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)). [ENOMEM]

Such a change would result in the break value being greater than or equal to the start address of any attached shared memory segment (see *shmop*(2)).

RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2).

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Path points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the path name. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chroot(2).

NAME

chmod – change mode of file

SYNOPSIS

```
int chmod (path, mode)
char *path;
int mode;
```

DESCRIPTION

Path points to a path name naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
02000	Set group ID on execution.
01000	Save text image after execution
00400	Read by owner
00200	Write by owner
00100	Execute (or search if a directory) by owner
00070	Read, write, execute (search) by group
00007	Read, write, execute (search) by others

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user or the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time.

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chown(2), mknod(2).

NAME

chown – change owner and group of a file

SYNOPSIS

int chown (path, owner, group)
char *path;
int owner, group;

DESCRIPTION

Path points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2).

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

Any component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not super-user. [EPERM]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chdir(2).

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*.

Close will fail if *fildes* is not a valid open file descriptor. [EBADF]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

```
int creat (path, mode)
char *path;
int mode;
```

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls. See *fcntl*(2). No process may have more than 80 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The path name is null. [ENOENT]

The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]

The named file resides or would reside on a read-only file system. [EROFS]

The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]

The file exists and write permission is denied. [EACCES]

The named file is an existing directory. [EISDIR]

Eighty (80) file descriptors are currently open. [EMFILE]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative integer, namely the

file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), dup(2), lseek(2), open(2), read(2), umask(2), write(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer. (i.e., both file descriptors share one file pointer.)

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Eighty (80) file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

```
int execl (path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, 0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header (see *a.out*(4)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Path points to a path name that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the calling process's environment in the global cell:

extern char **environ;

and it is used to pass the calling process's environment to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Except for SIGPHONE and SIGWIND, signals set to be caught by the calling process will be set to terminate new process; see *signal*(2).

If the set-user-ID mode bit of the new process file is set (see *chmod*(2)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process will not be attached to the new process (see *shmop*(2)).

Profiling is disabled for the new process; see *profil*(2).

The new process also inherits the following attributes from the calling process:

nice value (see *nice*(2))
process ID
parent process ID
process group ID
semadj values (see *semop*(2))
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))

Exec will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's path name do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

The new process requires more memory than is allowed by the system-imposed maximum MAXMEM. [ENOMEM]

The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

Path, *argv*, or *envp* point to an illegal address. [EFAULT]

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

SEE ALSO

exit(2), fork(2), environ(5).

NAME

exit, _exit – terminate process

SYNOPSIS

void exit (status)
int status;
void _exit (status)
int status;

DESCRIPTION

Exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see <**sys/proc.h**>) to be used by *times*.

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(2)) inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value (see *semop*(2)), that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock* (2)].

If the process ID, tty group ID, and process group ID of the calling process are equal, (i.e. it is a process group leader), the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

If the process is a process group leader, all processes in its group are made members of the *null* group.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

intro(2), semop(2), signal(2), wait(2).

EXIT (2)

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The *cmd*s available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL Get *file* status flags.

F_SETFL Set *file* status flags to *arg*. Only certain flags can be set; see *fcntl*(5).

F_GETLK Get the first block which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(5)]. The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return

immediately with an error value of −1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure describes the type (*l_type*), starting offset (*l_start*), relative offset (*l_whence*), size (*l_len*), process id (*l_pid*), and system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of the file by setting *l_len* to zero (0). If such a lock also has *l_where* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file [see *chmod*(2)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

Fcntl will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EINVAL] *Cmd* is F_DUPFD. *Arg* is either negative, or greater than or equal to, the configured value for the maximum number of open file descriptors allowed each user.

[EINVAL] *Cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.

[EACCES] *Cmd* is F_SETLK, the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write lock (F_WRLCK) and the segment of a file to be locked is already read or write locked by another process.

[ENOLCK] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked).

[EDEADLK] *Cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling process to sleep, waiting for that lock to become free, would cause a deadlock.

[EFAULT] *Cmd* is F_SETLK, *arg* points outside the program address space.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than −1.
F_GETFL	Value of file flags.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.

WARNINGS

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

NAME

fork – create a new process

SYNOPSIS

int fork ( )

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., **SIG_DFL**, **SIG_ING**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value (see *nice*(2))
all attached shared memory segments (see *shmop*(2))
process group ID
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared (see *semop*(2)).

Process locks, text locks and data locks are not inherited by the child (see *plock*(2)).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0.

The child process has a different amount of time left until an alarm clock signal (see *alarm*(2)).

Fork will fail and no child process will be created if one or more of the following are true:

The system-imposed limit on the total number of processes under execution would be exceeded. [EAGAIN]

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. [EAGAIN]

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exec(2), times(2), wait(2).

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

int getpid ( )

int getpgrp ( )

int getppid ( )

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

int getuid ( )

int geteuid ( )

int getgid ( )

int getegid ( )

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl – control device

SYNOPSIS

ioctl (fildes, request, arg)

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in Section 7 discuss how *ioctl* applies to them.

Ioctl will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Fildes is not associated with a character special device. [ENOTTY]

Request or *arg* is not valid. See Section 7. [EINVAL]

RETURN VALUE

If an error has occurred, a value of −1 is returned and *errno* is set to indicate the error.

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(2)) and will be referred to below as *proc0* and *proc1* respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is −1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is −1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not −1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be sent if one or more of the following are true:

Sig is not a valid signal number. [EINVAL]

No process can be found corresponding to that specified by *pid*. [ESRCH]

The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2).

NAME

link – link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link will fail and no link will be created if one or more of the following are true:

A component of either path prefix is not a directory. [ENOTDIR]

A component of either path prefix does not exist. [ENOENT]

A component of either path prefix denies search permission. [EACCES]

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not super-user. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

Path2 points to a null path name. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

unlink(2).

NAME

lseek – move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

Whence is not 0, 1 or 2. [EINVAL and SIGSYS signal]

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

mknod – make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*, where the value of *mode* is interpreted as follows:

0170000 file type; one of the following:
- 0010000 fifo special
- 0020000 character special
- 0040000 directory
- 0060000 block special
- 0100000 or 0000000 ordinary file

0004000 set user ID on execution
0002000 set group ID on execution
0001000 save text image after execution
0000777 access permissions; constructed from the following
- 0000400 read by owner
- 0000200 write by owner
- 0000100 execute (search on directory) by owner
- 0000070 read, write, execute (search) by group
- 0000007 read, write, execute (search) by others

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2). If *mode* indicates a block or character special file, *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Mknod may be invoked only by the super-user for file types other than FIFO special.

Mknod will fail and the new file will not be created if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(1), chmod(2), exec(2), umask(2), fs(4).

NAME

mount – mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. is the standard UNIX PC directory for mounting floppy diskettes. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

Mount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

Dir is not a directory. [ENOTDIR]

Spec or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]

The device associated with *spec* is currently mounted. [EBUSY]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(1M), umount(2).

NAME

msgctl – message control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

DESCRIPTION

Msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*.

Msgctl will fail if one or more of the following are true:

Msqid is not a valid message queue identifier. [EINVAL]

Cmd is not a valid command. [EINVAL]

Cmd is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super user and it is not equal to the value of **msg_perm.uid** in the data structure associated with *msqid*. [EPERM]

Cmd is equal to **IPC_SET,** an attempt is being made to increase to the value of **msg_qbytes,** and the effective

user ID of the calling process is not equal to that of super user. [EPERM]

Buf points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgget(2), msgop(2), stdipc(3C).

NAME

msgget – get message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

DESCRIPTION

Msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure (see *intro*(2)) are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

Msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.

Msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.

Msg_ctime is set equal to the current time.

Msg_qbytes is set equal to the system limit.

Msgget will fail if one or more of the following are true:

A message queue identifier exists for *key* but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *msgflg* would not be granted. [EACCES]

A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false". [ENOENT]

A message queue identifier is to be created but the system imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded. [ENOSPC]

A message queue identifier exists for *key* but ((*msgflg* & **IPC_CREAT**) & (*msgflg* & **IPC_EXCL**)) is "true". [EEXIST]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgctl(2), msgop(2), stdipc(3C).

NAME

msgop – message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

Msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *Msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *Mtext* is any text of length *msgsz* bytes. *Msgsz* can range from 0 to a system imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes** (see *intro*(2)).

The total number of messages on all queues system wide is equal to the system imposed limit.

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message will not be sent and the calling process will return immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and

the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgsnd will fail and no message will be sent if one or more of the following are true:

Msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Mtype is less than 1. [EINVAL]

The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true". [EAGAIN]

Msgsz is less than zero or greater than the system imposed limit. [EINVAL]

Msgp points to an illegal address. [EFAULT]

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

Msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is the received message's type as specified by the sending process. *Mtext* is the text of the message. *Msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of −1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgrcv will fail and no message will be received if one or more of the following are true:

Msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process. [EACCES]

Msgsz is less than 0. [EINVAL]

Mtext is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false". [E2BIG]

The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true". [ENOMSG]

Msgp points to an illegal address. [EFAULT]

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

RETURN VALUES

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgctl(2), msgget(2), stdipc(3C).

NAME

nice – change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

Nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Nice will fail and not change the nice value if *incr* is negative and the effective user ID of the calling process is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

nice(1), exec(2).

NAME

open – open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag [ , mode ] )
char *path;
int oflag, mode;
```

DESCRIPTION

Path points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process's effective

user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

Upon successful completion a non-negative integer, the file descriptor, is returned.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

No process may have more than 80 file descriptors open simultaneously.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

Eighty (80) file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the process's allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2).

NAME

pause – suspend process until signal

SYNOPSIS

pause ( )

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal catching-function (see *signal*(2)), the calling process resumes execution from the point of suspension; with a return value of −1 from *pause* and *errno* set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Writes up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

Pipe will fail if 19 or more file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

sh(1), read(2), write(2).

NAME

plock – lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (op)
int op;

DESCRIPTION

Plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK – lock text and data segments into memory (process lock)

TXTLOCK – lock text segment into memory (text lock)

DATLOCK – lock data segment into memory (data lock)

UNLOCK – remove locks

Plock will fail and not perform the requested operation if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

Op is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process. [EINVAL]

Op is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process. [EINVAL]

Op is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process. [EINVAL]

Op is equal to **UNLOCK** and no type of lock exists on the calling process. [EINVAL]

RETURN VALUE

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2).

NAME

profil – execution time profile

SYNOPSIS

void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), monitor(3C).

BUGS

Profil() is not supported on the UNIX PC.

NAME

ptrace – process trace

SYNOPSIS

int ptrace (request, pid, addr, data);
int request, pid, addr, data;

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *sdb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated (as on PDP-11s), request **1** returns a word from I space, and request **2** returns a word from D space. If I and D space are not separated (as on the 3B-20 and VAX-11/780), either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses range from 0 to 1024. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. Request 4 writes a word into I space, and request 5 writes a word into D space. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of −1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are:

the general registers (D0-D7, A0-A7)

certain bits of the Processor Status Word (all bits except SUPERVISOR state and interrupt level)

the PC register

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of −1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request sets the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request **7**. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal **SIGTRAP**.

GENERAL ERRORS

Ptrace will in general fail if one or more of the following are true:

Request is an illegal number. [EIO]

Pid identifies a child that does not exist or has not executed a *ptrace* with request **0**. [ESRCH]

SEE ALSO

sdb(1), exec(2), signal(2), wait(2).

NAME

read – read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data becomes available.

Read will fail if one or more of the following are true:

Fildes is not a valid file descriptor open for reading. [EBADF]

Buf points outside the allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), open(2), pipe(2), termio(7), window(7).

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort array[ ];
} arg;
```

DESCRIPTION

Semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL Return the value of semval (see *intro*(2)). {READ}

SETVAL Set the value of semval to *arg.val*. {ALTER} When this cmd is successfully executed the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of sempid. {READ}

GETNCNT Return the value of semncnt. {READ}

GETZCNT Return the value of semzcnt. {READ}

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL Place semvals into array pointed to by *arg.array*. {READ}

SETALL Set semvals according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to

the corresponding value found in the structure pointed to by *arg.buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This command can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

Semctl will fail if one or more of the following are true:

Semid is not a valid semaphore identifier. [EINVAL]

Semnum is less than zero or greater than **sem_nsems**. [EINVAL]

Cmd is not a valid command. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum. [ERANGE]

Cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super user and it is not equal to the value of **sem_perm.uid** in the data structure associated with *semid*. [EPERM]

Arg.buf points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval.
GETPID	The value of sempid.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
All others	A value of 0.

Otherwise, a value of **−1** is returned and *errno* is set to indicate the error.

SEE ALSO

semget(2), semop(2), stdipc(3C).

NAME

semget – get set of semaphores

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

Semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores (see *intro*(2)) are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

Semget will fail if one or more of the following are true:

Nsems is either less than or equal to zero or greater than the system imposed limit. [EINVAL]

A semaphore identifier exists for *key* but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *semflg* would not be granted. [EACCES]

A semaphore identifier exists for *key* but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero. [EINVAL]

A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is "false". [ENOENT]

A semaphore identifier is to be created but the system imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded. [ENOSPC]

A semaphore identifier is to be created but the system imposed limit on the maximum number of allowed semaphores system wide would be exceeded. [ENOSPC]

A semaphore identifier exists for *key* but ((*semflg* & **IPC_CREAT**) & (*semflg* & **IPC_EXCL**)) is "true". [EEXIST]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

semctl(2), semop(2), stdipc(3C).

NAME

semop – semaphore operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf (*sops)[];
int nsops;
```

DESCRIPTION

Semop is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *Sops* is a pointer to the array of semaphore-operation structures. *Nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

Sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If semval (see *intro*(2)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value (see *exit*(2)) for the specified semaphore.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling

process's semadj value for the specified semaphore.

The semid for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If semval is zero, *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The semid for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

Semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

Semid is not a valid semaphore identifier. [EINVAL]

Sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*. [EFBIG]

Nsops is greater than the system imposed maximum. [E2BIG]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true". [EAGAIN]

The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded. [ENOSPC]

The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit. [EINVAL]

An operation would cause a semval to overflow the system imposed limit. [ERANGE]

An operation would cause a semadj value to overflow the system imposed limit. [ERANGE]

Sops points to an illegal address. [EFAULT]

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, the value of semval at the time of the call for the last operation in the array pointed to by *sops* is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), semctl(2), semget(2), stdipc(3C).

NAME

setpgrp – set process group ID

SYNOPSIS

int setpgrp ( )

DESCRIPTION

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2), window(7).

BUGS

Setpgrp cannot be called from processes associated with windows. Any process calling *setpgrp* must have stdin, stdout, and stderr directed to devices other than window devices to function properly.

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

Setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

getuid(2), intro(2).

NAME

shmctl – shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

DESCRIPTION

Shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

Shmctl will fail if one or more of the following are true:

Shmid is not a valid shared memory identifier. [EINVAL]

Cmd is not a valid command. [EINVAL]

Cmd is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal

to that of super user and it is not equal to the value of **shm_perm.uid** in the data structure associated with *shmid*. [EPERM]

Buf points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

shmget(2), shmop(2), stdipc(3C).

NAME

shmget – get shared memory segment

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

DESCRIPTION

Shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see *intro*(2)) are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE.**

Key does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

Shm_perm.cuid, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **Shm_segsz** is set equal to the value of *size*.

Shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

Shm_ctime is set equal to the current time.

Shmget will fail if one or more of the following are true:

Size is less than the system imposed minimum or greater than the system imposed maximum. [EINVAL]

A shared memory identifier exists for *key* but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *shmflg* would not be granted. [EACCES]

A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero. [EINVAL]

A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false". [ENOENT]

A shared memory identifier is to be created but the system imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded. [ENOSPC]

A shared memory identifier and associated shared memory segment are to be created but the amount of available

physical memory is not sufficient to fill the request. [ENOMEM]

A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) & (*shmflg* & **IPC_EXCL**)) is "true". [EEXIST]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a shared memory identifier, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

shmctl(2), shmop(2), stdipc(3C).

NAME

shmop – shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr
```

DESCRIPTION

Shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "false", the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if (*shmflg* & SHM_RDONLY) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

Shmat will fail and not attach the shared memory segment if one or more of the following are true:

Shmid is not a valid shared memory identifier. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

The available data space is not large enough to accommodate the shared memory segment. [ENOMEM]

Shmaddr is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address. [EINVAL]

Shmaddr is not equal to zero, (*shmflg* & SHM_RND) is "false", and the value of *shmaddr* is an illegal address. [EINVAL]

The number of shared memory segments attached to the calling process would exceed the system imposed limit. [EMFILE]

Shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

Shmdt will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment. [EINVAL]

RETURN VALUES

Upon successful completion, the return value is as follows:

Shmat returns the data segment start address of the attached shared memory segment.

Shmdt returns a value of 0.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), shmctl(2), shmget(2), stdipc(3C).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <sys/signal.h>

int (*signal (sig, func))( )
int sig;
int (*func)( );
```

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	death of a child (see *WARNING* below)
SIGPWR	19	power fail (see *WARNING* below)
SIGWIND	20	window status changes
SIGPHONE	21	telephone status changes

See below for the significance of the asterisk (*) in the above list.

Func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2) plus a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signal **SIGKILL** cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call will return a –1 to the calling process with *errno* set to EINTR.

Note: the signal **SIGKILL** cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending **SIGKILL** signal.

Signal will fail if one or more of the following are true:

Sig is an illegal signal number, including **SIGKILL**. [EINVAL]

Func points to an illegal address. [EFAULT]

SIGWIND and SIGPHONE are ignored by default and are reset to SIG.IGN upon an *exec*(2) system call.

RETURN VALUE

Upon successful completion, ***signal*** returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C).

WARNING

Two other signals that behave differently than the signals described above exist in this release of the system; they are:

SIGCLD	18	death of a child (reset when caught)
SIGPWR	19	power fail (not reset when caught)

There is no guarantee that, in future releases of UNIX, these signals will continue to behave as described below; they are included only for compatibility with other versions of UNIX. Their use in new programs is strongly discouraged.

For these signals, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate; see *exit*(2).

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** except, that while the process is executing the signal-catching function any received **SIGCLD** signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

The **SIGCLD** affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Path points to a path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
dev_t    st_dev;    /* ID of device containing */
                    /* a directory entry for this file */
ino_t    st_ino;    /* Inode number */
ushort   st_mode;   /* File mode; see mknod(2) */
short    st_nlink;  /* Number of links */
ushort   st_uid;    /* User ID of the file's owner */
ushort   st_gid;    /* Group ID of the file's group */
dev_t    st_rdev;   /* ID of device */
                    /* This entry is defined only for */
                    /* character special or block */
                    /* special files */
off_t    st_size;   /* File size in bytes */
time_t   st_atime;  /* Time of last access */
time_t   st_mtime;  /* Time of last data modification */
time_t   st_ctime;  /* Time of last file status change */
                    /* Times measured in seconds */
                    /* since 00:00:00 GMT, */
                    /* Jan. 1, 1970 */
```

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Stat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Buf or *path* points to an invalid address. [EFAULT]

Fstat will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Buf points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), time(2), unlink(2).

NAME

stime – set time

SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Stime will fail if the effective user ID of the calling process is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(2).

NAME

sync – update super-block

SYNOPSIS

void sync ( )

DESCRIPTION

Sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

Syslocal – local system calls

SYNOPSIS

#include <sys/syslocal.h>

int syslocal (cmd [, arg] ...)

DESCRIPTION

Syslocal executes special AT&T UNIX PC system calls. *Cmd* is the name of one of the system calls described below.

SYSL_REBOOT — Reboots the system. You must be superuser to execute. No additional arguments are required.

SYSL_KADDR — Returns certain kernel addresses or values. This call is used by programs like *ps*(1) so that they don't have to read the kernel symbol table. The second argument is one of the following:

SLA_V	returns address of V
SLA_PROC	returns address of proc table
SLA_TIME	returns address of system time
SLA_USRSTK	returns top of user stack
SLA_USIGN	returns signature, unique # for each version
SLA_BLDDATE	returns address of build date string
SLA_BLDPWD	returns address of build directory string
SLA_MEM	returns size of physical memory
SLA_BDEVCNT	returns maximum number of block devices
SLA_CDEVCNT	returns maximum number of character devices

SYSL_LED — Turns on/off user LED. The second argument is either 0 for off or 1 for on.

The following two calls support the hardware real-time clock. Their use requires the additional include file:

#include <sys/rtc.h>

SYSL_RDRTC — Reads the real-time clock. The second argument is a *struct rtc* * .

SYSL_WRTRTC — Writes the real-time clock. The second argument is a *struct rtc* * .

The following two calls support loadable device drivers. Their use requires the additional include file:

#include <sys/drv.h>

SYSL_ALLOCDRV Allocates/deallocates space for a loadable driver and returns driver status. The second argument is one of the following:

DRVALLOC	allocates space
DRVUNALLOC	releases allocated space
DRVSTAT	returns driver status

The third argument is a *struct drvalloc ** You must be superuser to execute DRVALLOC and DRVUNALLOC.

SYSL_BINDDRV Loads/unloads a loadable driver. The second argument is either DRVBIND for loading or DRVUNBIND for unloading. The third argument is a *struct drvbind ** . You must be superuser to execute.

The following two calls support installable fonts.

SYSL_LFONT Installs a font.

SYSL_UFONT Deinstalls a font.

In both cases, two arguments are required: the font file pathname (dummy pointer for SYSL_UFONT) and the font slot number (0 to 15). Again, you must be superuser to execute. See *window*(7) for additional font information.

Supplying a font slot number between 0 and 7 causes the font to be inherited at that slot number by all subsequent windows. Preloading fonts into slots 8-15 allows these fonts to be installed without going to the file system so they can be loaded rapidly. This is useful for applications which refer to more than 8 fonts because the font activity is more efficient.

If you attempt to load a font into a slot which is currently occupied, you will not get an error condition, but rather, the old font will be swapped out and the new one loaded in. You can also deinstall a font from slots 0 through 7, if the font to be deinstalled is not being accessed. If it is being accessed ERRNO is set to EBUSY.

NAME

time – get time

SYNOPSIS

```
long time ((long *) 0)

long time (tloc)
long *tloc;
```

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

Time will fail if *tloc* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

stime(2).

NAME

times – get process and child process times

SYNOPSIS

#include <sys/types.h>
#include <sys/times.h>

long times (buffer)
struct tms *buffer;

DESCRIPTION

Times fills the structure pointed to by *buffer* with time-accounting information. The following is this contents of the structure:

```
struct   tms {
         time_t   tms_utime;
         time_t   tms_stime;
         time_t   tms_cutime;
         time_t   tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are in 60ths of a second on DEC processors, 100ths of a second on WECo processors.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

Tms_cstime is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

Times will fail if *buffer* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths (100ths) of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM]

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

brk(2), write(2).

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1), chmod(2), creat(2), mknod(2), open(2).

NAME

umount – unmount a file system

SYNOPSIS

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

Umount will fail if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

Spec does not exist. [ENXIO]

Spec is not a block special device. [ENOTBLK]

Spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

Spec points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(2).

NAME

uname – get name of current UNIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current UNIX system in the structure pointed to by *name*.

Uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

Uname returns a null-terminated character string naming the current UNIX system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that UNIX is running on.

Uname will fail if *name* points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

uname(1).

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to be *path*.

The named file is unlinked unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Write permission is denied on the directory containing the link to be removed. [EACCES]

The named file is a directory and the effective user ID of the process is not super-user. [EPERM]

The entry to be unlinked is the mount point for a mounted file system. [EBUSY]

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1), close(2), link(2), open(2).

NAME

ustat – get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes to following elements:

```
daddr_t  f_tfree;      /* Total free blocks */
ino_t    f_tinode;     /* Number of free inodes */
char     f_fname[6];   /* Filsys name */
char     f_fpack[6];   /* Filsys pack name */
```

Ustat will fail if one or more of the following are true:

Dev is not the device number of a device containing a mounted file system. [EINVAL]

Buf points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2), fs(4).

NAME

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

Path points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is **NULL**, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not **NULL**, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct   utimbuf {
         time_t   actime;      /* access time */
         time_t   modtime;     /* modification time */
};
```

Utime will fail if one or more of the following are true:

The named file does not exist. [ENOENT]

A component of the path prefix is not a directory. [ENOTDIR]

Search permission is denied by a component of the path prefix. [EACCES]

The effective user ID is not super-user and not the owner of the file and *times* is not **NULL**. [EPERM]

The effective user ID is not super-user and not the owner of the file and *times* is **NULL** and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

Times is not **NULL** and points outside the process's allocated address space. [EFAULT]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2).

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

DESCRIPTION

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(2)), or until any one of the calling process's child processes stops in a trace mode (see *ptrace*(2)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.

If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit*; see *exit*(2).

If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal*(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see *intro*(2).

Wait will fail and return immediately if one or more of the following are true:

The calling process has no existing unwaited-for child processes. [ECHILD]

Stat_loc points to an illegal address. [EFAULT]

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), pause(2), signal(2).

WARNING

See *WARNING* in *signal*(2).

NAME

write – write on a file

SYNOPSIS

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

Fildes is not a valid file descriptor open for writing. [EBADF]

An attempt is made to write to a pipe that is not open for reading by any process. [EPIPE and SIGPIPE signal]

An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2). [EFBIG]

Buf points outside the process's allocated address space. [EFAULT]

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes will be permitted. Thus, the write will fail if a write of *nbyte* bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3M) These functions constitute the Math Library, *libm*. They are automatically loaded as needed by the FORTRAN compiler. They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**.

(3T) These functions constitute the UNIX PC "terminal access method" (tam) library.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include his own definition if he is not using **<stdio.h>**.

FILES

/lib/libc.a
/lib/libm.a

SEE ALSO

ar(1), cc(1), ld(1), nm(1), intro(2), stdio(3S).

DIAGNOSTICS

Functions in the Math Library (3M) may return the conventional values **0** or **HUGE** (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE. As many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

L64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – generate an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

Abort causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO

adb(1), exit(2), kill(2), signal(2).

DIAGNOSTICS

If **SIGIOT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "abort – core dumped" is written by the shell.

NAME

abs – return integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

BUGS

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

ASSERT (3X)

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **–DNDEBUG** (see *cpp* (1)), or with the preprocessor control statement "#define NDEBUG" ahead of the "#include <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

atof – convert ASCII string to floating-point number

SYNOPSIS

double atof (nptr)
char *nptr;

DESCRIPTION

Atof converts a character string pointed to by *nptr* to a double-precision floating-point number. The first unrecognized character ends the conversion. *Atof* recognizes an optional string of white-space characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer. If the string begins with an unrecognized character, *atof* returns the value zero.

DIAGNOSTICS

When the correct value would overflow, *atof* returns **HUGE**, and sets *errno* to **ERANGE**. Zero is returned on underflow.

SEE ALSO

scanf(3S).

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

J0 and *j1* return Bessel functions of x of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of x of the first kind of order n.

Y0 and *y1* return the Bessel functions of x of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of x of the second kind of order n. The value of x must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **HUGE** and to set *errno* to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output; the process will continue.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

bsearch – binary search

SYNOPSIS

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to the datum to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

lsearch(3C), hsearch(3C), qsort(3C), tsearch(3C).

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S).

The resolution of the clock is 16.667 milliseconds.

SEE ALSO

times(2), wait(2), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from −1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S).

NAME

crypt, setkey, encrypt – generate DES encryption

SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, edflag)
char *block;
int edflag;
```

DESCRIPTION

This function is available only in the domestic (U.S.) version of the UNIX PC software.

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

Key is a user's typed password. *Salt* is a two-character string chosen from the set [**a-z A-Z 0-9 . /**]; this string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the above mentioned algorithm to encrypt or decrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

login(1M), passwd(1), getpass(3C), passwd(4).

BUGS

The return value points to static data that are overwritten by each call.

CTERMID (3S)

NAME

ctermid – generate file name for terminal

SYNOPSIS

```
#include <stdio.h>

char *ctermid(s)
char *s;
```

DESCRIPTION

Ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the **<stdio.h>** header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to string

SYNOPSIS

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

Sun Sep 16 01:03:52 1973\n\0

Localtime and *gmtime* return pointers to "tm" structures, described below. *Localtime* corrects for the time zone and possible Daylight Savings Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the UNIX system uses.

Asctime converts a "tm" structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the "tm" structure, are in the *<time.h>* header file. The structure declaration is:

```
struct tm {
        int  tm_sec;    /* seconds (0 - 59) */
        int  tm_min;    /* minutes (0 - 59) */
        int  tm_hour;   /* hours (0 - 23) */
        int  tm_mday;   /* day of month (1 - 31) */
        int  tm_mon;    /* month of year (0 - 11) */
        int  tm_year;   /* year - 1900 */
        int  tm_wday;   /* day of week (Sunday = 0) */
        int  tm_yday;   /* day of year (0 - 365) */
        int  tm_isdst;
};
```

Tm_isdst is non-zero if Daylight Savings Time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named **TZ** is present, *asctime* uses the contents of the variable to override the default time zone. The value of **TZ** must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting **TZ** are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

char *tzname[2] = { "EST", "EDT" };

are set from the environment variable **TZ**. The function *tzset* sets these external variables from **TZ**; *tzset* is called by *asctime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set by default when the user logs on, to a value in the local **/etc/profile** file (see *profile*(4)).

SEE ALSO

time(2), getenv(3C), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

. . .

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** (−1 – see *stdio*(3S)).

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

ascii(5).

NAME

curses – screen functions with "optimal" cursor motion

SYNOPSIS

cc [flags] files **–lcurses –ltermcap** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold,
termio(7) termcap(5)

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) coordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping

noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	change all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

 CUSERID (3S)

NAME

cuserid – get character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

DESCRIPTION

Cuserid generates a character-string representation of the login name of the owner of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character **(\0)** will be placed at *s[0]*.

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

DESCRIPTION

Dial returns a file-descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the **<dial.h>** header file.

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The CALL typedef in the **<dial.h>** header file is:

```
typedef struct {
        struct termio *attr;    /* pointer to termio */
                                /* attribute struct */
        int         baud;       /* transmission data rate */
        int         speed;      /* 212A modem: low=300, */
                                /* high=1200 */
        char        *line;      /* device name for */
                                /* out-going line */
        char        *telno;     /* pointer to tel-no */
                                /* digits string */
        int         modem;      /* specify modem control */
                                /* for direct lines */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high or low speed setting on the 212A modem. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the **L-devices** file. In this case, the value of the *baud* element need not be specified as it will be determined from the **L-devices** file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described in *phone*(7). The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is

required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the **<termio.h>** header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2).
phone(7), termio(7) in the *UNIX Administrator's Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the **<dial.h>** header file.

```
INTRPT   -1  /* interrupt occurred */
D_HUNG   -2  /* dialer hung (no return from write) */
NO_ANS   -3  /* no answer within 10 seconds */
ILL_BD   -4  /* illegal baud-rate */
A_PROB   -5  /* acu problem (open() failure) */
L_PROB   -6  /* line problem (open() failure) */
NO_Ldv   -7  /* can't open LDEVS file */
DV_NT_A  -8  /* requested device not available */
DV_NT_K  -9  /* requested device not known */
NO_BD_A  -10 /* no device available at requested baud */
NO_BD_K  -11 /* no device known at requested baud */
```

WARNINGS

Including the **<dial.h>** header file automatically includes the **<termio.h> header file.**

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$, hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

$a = 5DEECE66D_{16} = 273673163155_8$
$c = B_{16} = 13_8$.

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48,* and a pointer to this buffer is the value returned by *seed48.* This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier a, and *param[6]* specifies the 16-bit addend c. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

NOTES

The versions of these routines for the VAX-11 and PDP-11 are coded in assembly language for maximum speed. It requires approximately 80 μsec on a VAX-11/780 and 130 μsec on a PDP-11/70 to generate one pseudo-random number. On other computers, the routines are coded in portable C. The source code for the portable version can even be used on computers which do not

DRAND48 (3C)

have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* do not exist; instead, they are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval $[0, m-1]$.

SEE ALSO

rand(3C).

NAME

ecvt, fcvt, gcvt – convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for FORTRAN F-format output of the number of digits specified by *ndigit*.

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The return values point to static data whose content is overwritten by each call.

END (3C)

NAME

end, etext, edata – last locations in program

SYNOPSIS

```
extern end;
extern etext;
extern edata;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**-p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

SEE ALSO

brk(2), malloc(3C), stdio(3S).

NAME

eprintf – send a message to the status manager

SYNOPSIS

```
#include <status.h>
int eprintf (mtype, mact, uname, format [, arg ] ... )
int mtype, mact;
char *uname, *format;
```

DESCRIPTION

Eprintf formats the passed message a la *printf* and writes the message to the error device. The status manager wakes up whenever the error device is written to, queues the message, and displays an icon to indicate a message is waiting.

Mtype (message type) can have one of the following values:

ST_MAIL	Mail messages
ST_CAL	Calendar messages
ST_OTHER	Miscellaneous messages
ST_SYS	Kernel error messages
ST_LOG	Log message in log file
ST_POP	Popup message

Mact (message action) can have one of the following values:

ST_DISPLAY	Just display message
ST_EXEC	Execute process (message text is shell command line in this case)
ST_NOTIFY	Notify caller on display (sends caller SIGUSR1)
ST_CONFIRM	Signal caller with confirmation/denial on display (SIGUSR1 = Yes, SIGUSR2 = No)
ST_OFF	Remove messages from queue
ST_LOGFILE	Log message in log file

Uname points to the user login name that the message is for. The status manager will only display the message pending icon when this user is logged in. If *uname* is NULL (or if it points to a null string), then the message is displayed regardless of who is logged in.

ST_POP will cause the message to be acted on immediately, rather than displaying an icon and waiting for the user to click. ST_LOG will take the first word of the formatted message (i.e., up to the first space) as a file name, which it will open as a logfile in **/usr/adm**. The rest of the message will then be inserted in the file, followed by a time stamp.

DIAGNOSTICS

Eprintf returns –1 if error (open of error device failed).

SEE ALSO

message (3T), tam(3T).

 ERF (3M)

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

Erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Erfc, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g. for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

DESCRIPTION

Exp returns e^x.

Log returns the natural logarithm of x. The value of x must be positive.

Log10 returns the logarithm base ten of x. The value of x must be positive.

Pow returns x^y. The values of x and y may not both be zero. If x is non-positive, y must be an integer.

Sqrt returns the square root of x. The value of x may not be negative.

DIAGNOSTICS

Exp returns **HUGE** when the correct value would overflow, and sets *errno* to **ERANGE**.

Log and *log10* return 0 and set *errno* to **EDOM** when x is non-positive. An error message is printed on the standard error output.

Pow returns 0 and sets *errno* to **EDOM** when x is non-positive and y is not an integer, or when x and y are both zero. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow, *pow* returns **HUGE** and sets *errno* to **ERANGE.**

Sqrt returns 0 and sets *errno* to **EDOM** when x is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

Fclose is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof (stream)
FILE
*stream;

int ferror (stream)
FILE
*stream;

void clearerr (stream)
FILE
*stream;

int fileno(stream)
FILE
*stream;
```

DESCRIPTION

Feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero.

Clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

Fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns x if y is zero, otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

Fabs returns $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by *file-name* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

File-name points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout** and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor obtained from *open*, *dup*, *creat*, or *pipe*(2), which will open files but not return pointers to a FILE structure *stream* which are necessary input for many of the section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek*

may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer on failure.

NAME

form – display and accept forms

SYNOPSIS

```
#include <menu.h>
#include <form.h>
int form(form, op)
form_t *form;
int op;
```

DESCRIPTION

This routine manipulates a form as determined by the operation code (*op*). If the *op* arg is F_BEGIN, the form is initialized and displayed. If *op* is F_INPUT, user input is accepted. If *op* is F_END, the form is terminated and removed from the display. These functions may be combined in many ways. By specifying (F_BEGIN | F_INPUT | F_END), the caller creates a "pop-up" form which is initialized (displayed), used for input, then removed. Generally, (F_BEGIN | F_INPUT) is used for the first call, F_INPUT for each subsequent interaction, and F_END when the form is to be discarded.

During the F_INPUT function, the user may point to fields with the mouse or with the keyboard (arrows, Prev, Next, Beg, Home, End, Tab). The user may may modify fields by typing and editing (Back Space, Dlete Char, Clear Line, Cancel) or by selecting a choice from a menu optionally associated with the field.

The *form* structure has the following form:

```
typedef struct
{
        char        *f_label;     /* form label */
        char        *f_name;      /* form name */
        char        f_flags;      /* form flags */
        int         f_win;        /* form window */
        track_t     *f_track;     /* tracking info */
        field_t     *f_fields;    /* fields */
        field_t     *f_curfl;     /* current field */
} form_t;
```

F_label is the form label, displayed on the window label line of the form. If *f_label* is NULL, no label is displayed.

F_name is the form name, or NULL if the form has no name.

F_flags contains flags. The F_WINNEW flag causes *form* to use the "new" algorithm to place the window. Basically, the new algorithm looks for relatively empty screen space to place the window. F_WINSON causes *form* to use the "son" algorithm which causes the new window to slightly overlap the current window. If neither F_WINNEW nor F_WINSON is given, the "popup" algorithm is used. This causes the new window to appear near the middle of the current window, inside it if possible. F_NOMOVE is set if the

Move icon is not to be displayed on the border of the form. F_NOHELP is set if the Help icon is not to be displayed on the form border.

F_win holds the window identifier associated with this form. It is allocated on an F_BEGIN call, used on subsequent calls, and deleted on an F_END call. *F_track* is a pointer to the mouse-tracking information required during form interaction. The space for this data is allocated on F_BEGIN and freed on F_END.

F_fields points to the array of fields (see below). *F_curfl* points to the current field. The caller should point *f_curfl* to the default field. *Form* will modify *f_curfl* as the user moves the highlighting around in the form. The list of fields is terminated by a field whose *fl_name* is NULL.

Each field in the array pointed to by *f_fields* and *f_curcl* has the following form:

```
typedef struct
{
        char        *fl_name;    /* field name */
        char        fl_row;      /* field row  */
        char        fl_ncol;     /* name column */
        char        fl_fcol;     /* field column */
        char        fl_len;      /* field length */
        char        fl_flags;    /* flags */
        char        *fl_value;   /* field values */
        menu_t      *fl_menu;    /* assoc. menu pointer */
        char        *fl_prompt;  /* field prompt */
} field_t;
```

Fl_name is the field name. *Fl_row* is the row number on which to display the field. Row (and column) numbers are form-relative with 0,0 being the upper-leftmost location in the form. The form name (*f_name*) is located above 0,0 so the user needn't allocate a row for it.

Fl_ncol and *fl_fcol* control where the field name (*fl_ncol*) and field value (*fl_fcol*) are displayed. Generally, *fl_fcol* is greater than *fl_ncol* by at least the length of the field name.

Fl_len is the length of the field. See *fl_value*, below.

Fl_flags contains various flags which describe the field. F_CLEARIT specifies that any previous value for the field should be erased when the user tries to enter a new value. This is useful for fields where user editing makes little sense. F_MONLY means that the only allowable input to this field is via the associated menu (see *fl_menu*, below).

On call, *fl_value* contains the initial field value. On return, this string is modified to contain the user-supplied value. If no editing was performed by the user, the return value is the same as the call value. Note that the caller must supply a pointer to a character array at least *fl_len* + 1 bytes long. In addition, the caller should place a null byte after the end of the default value. For a 30 byte

field, a default value might be of the form:

```
"Default Value\0              "
 1234567890123 45678901234567890
       1       2       3
```

Fl_menu points to an optional "associated menu." If the caller supplies a menu pointer, then the user may press the Cmd or Opts key on that field to invoke *menu*(3T) to parse the menu. The selected menu item's name (*mi_name*) is placed in the field's value (*fl_value*). If the F_MONLY flag is set for the field, then any attempt to edit the field's value will force the associated menu to pop-up. When a field has an associated menu, the SLECT and MARK keys step through the menu choices without displaying the menu.

The optional message pointed to by *fl_prompt* is displayed on the prompt line whenever the field is selected. As the user moves from field to field, the prompt changes.

EXAMPLE

The following program illustrates a typical use of *form*:

```
#include <tam.h>
#include <menu.h>
#include <form.h>
#include <stdio.h>
#include <kcodes.h>

mitem_t printitems[ ] =
{
        "ASR-33",               0,0,
        "Centronix",            0,1,
        "Diablo #1",            0,2,
        "Diablo #2",            0,3,
        "Epson in lab",         0,4,
        "Laser Printer",        0,5,
        "File",                 0,6,
        0,                      0,0
};

menu_t printmenu =
{
         "Printers",
         0,
         "Select a Printer from the list",
         0,1,0,0,
         M_SINGLE,
         {0},
         0,0,0,0,0,
         printitems,
         printitems,
         0
};

mitem_t priitems[ ] =
```

```
{
        "Low",          0,0
        "Normal",       0,1,
        "High",         0,2,
        "Immediate",    0,3,
        0,0,0,
};

menu_t primenu =
{
        "Printing Priority",
        0,
        "At what priority should the document be printed?",
        0,1,0,0,
        M_SINGLE,
        {0},
        0,0,0,0,0,
        priitems,
        &priitems[1],
        0
};

mitem_t yesnoitems[ ] =
{
        "No",   0,0,
        "Yes",  0,1,
        0,0,0
};

menu_t yesnomenu =
{
        0,
        0, "Select Yes (y) or No (n)",
        0,1,0,0,
        M_SINGLE,
        {0},
        0,0,0,0,0,
        yesnoitems,
        yesnoitems,
        0

};

field_t printfields [ ] =
{
        "Printer Name", 0,0,15,30,F_CLEARIT,
        "System Printer                ",&printmenu,
        "Enter a Printer Name (touch CMD or OPTS to see
        choices)",
        "From Page",    1,0,15,5,0,
        "1      ",0,
        "Select the page number of the first page to be printed",
```

```
        "To Page",      1,25,40,5,0,
        "999    ",0,
        "Select the page number of the last page to be printed",

        "Priority",     2,0,15,10,F_MONLY,
        "Normal   ",&primenu,
        "Enter the print priority (Press CMD or OPTS to see
        choices)",

        "Delete After Printing?",       4,0,25,3,0,
        "No   ", &yesnomenu,
        "Do you wish the document to be deleted after it is
        printed?",

        0,                              0,0,0,0,0,
                0,0,
                0
};

form_t printform =
{
        "Print",
        "Printer Options",
        0,
        0,
        0,
        printfields,
        printfields
};



main()
{
        int err;
        int printop;
        char *which;

        winit();
        keypad(0,1);

        printop = M_BEGIN | M_INPUT;

        while(1)
        {
                which = "printform";
                err = form( &printform, printop );
                printop &= ~M_BEGIN;
                if ( err < 0 || err == Close )
                break;
        }
```

```
        if ( err < 0 )
        {
                fprintf(stderr,"fatal err in %s, code =
                %d",which,err);
                sleep(5);
        }
        wexit(0);
}
```

FILES

/usr/include/form.h
/usr/include/menu.h
/usr/include/kcodes.h

SEE ALSO

menu(3T), tam(3T).

DIAGNOSTICS

Form returns non-negative keyboard codes (see **kcodes.h**) when keyboard input terminated the form interaction. Other return values signal more serious errors and are defined in **form.h**.

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Fread copies, into an array beginning at *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The variable *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Fread and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq | x | < 1.0$, and the "exponent" n is an integer. *Frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*.

Ldexp returns the quantity *value* $* 2^{exp}$.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, **HUGE** is returned and *errno* is set to **ERANGE**.

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although in UNIX an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such a offset, which is not necessarily measured in bytes.

NAME

ftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containing information about the object, and an integer. Possible values of the integer, defined in the **<ftw.h>** header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns –1, and sets the error type in *errno*.

Ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *Ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
Ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

 GAMMA (3M)

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

extern int signgam;

double gamma (x)
double x;

DESCRIPTION

Gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-negative integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LOGHUGE)
        error( );
y = signgam * exp(y);
```

where LOGHUGE is the least value that causes *exp*(3M) to return a range error.

DIAGNOSTICS

For non-negative integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating DOMAIN error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

exp(3M), matherr(3M).

NAME

getc, getchar, fgetc, getw – get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e. byte) from the named input *stream*. It also moves the file pointer, if defined, ahead one character in *stream*. *Getc* is a macro and so cannot be used if a function is necessary; for example one cannot have a function pointer point to it.

Getchar returns the next character from the standard input stream, *stdin*. As in the case of *getc*, *getchar* is a macro.

Fgetc performs the same function as *getc*, but is a genuine function. *Fgetc* runs more slowly than *getc*, but takes less space per invocation.

Getw returns the next word (i.e. integer) from the named input *stream*. The size of a word varies from machine to machine. It returns the constant **EOF** upon end-of-file or error, but as that is a valid integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *Getw* increments the associated file pointer, if defined, to point to the next word. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the integer constant **EOF** at end-of-file or upon an error.

BUGS

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** doesn't work sensibly. *Fgetc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

DESCRIPTION

Getcwd returns a pointer to the current directory path-name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer.

SEE ALSO

environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )
```

DESCRIPTION

Getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a "group" structure, defined in the <*grp.h*> header file.

```
struct group {
        char    *gr_name;     /* the name of the group */
        char    *gr_passwd;   /* the encrypted group */
                              /* password */
        int     gr_gid;       /* the numerical group ID */
        char    **gr_mem;     /* vector of pointers to */
                              /* member names */
};
```

Getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group ID matching *gid* is found and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

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

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

Getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Returns the **NULL** pointer if *name* not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option -- may be used to delimit the end of the options; **EOF** will be returned, and -- will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
```

```
                        errflg++;
                    else
                        aflg++;
                    break;
                case 'b':
                    if (aflg)
                        errflg++;
                    else
                        bproc( );
                    break;
                case 'f':
                    ifile = optarg;
                    break;
                case 'o':
                    ofile = optarg;
                    bufsiza = 512;
                    break;
                case '?':
                    errflg++;
                }
            if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (2);
            }
            for ( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
            :
            :
        }
```

SEE ALSO

getopt(1).

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

Getpass reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string ***prompt*** and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpent, endpent – get and clean up printer status file entries

SYNOPSIS

```
#include <lp.h>

int getpent(p)
struct pstat *p;
int endpent()
```

DESCRIPTION

Getpent returns a structure describing a printer that is installed in the *lp* spooler subsystem. EOF is returned when no more printers are available.

Endpent is used to clean up after the last call to getpent.

```
struct pstat                            /* printer status entry */
{
    char    p_dest[DESTMAX+1];  /* destination name of printer */
    int     p_pid;              /* if busy, process id that is */
                                /* printing, otherwise 0 */
    char    p_rdest[DESTMAX+1];/* if busy, the destination */
                                /* requested by user at time of */
                                /* request, otherwise "-" */
    int     p_seqno;            /* if busy, sequence # of */
                                /* printing request */
    time_t  p_date;             /* date last enabled/disabled */
    char    p_reason[P_RSIZE];  /* if enabled, then "enabled" */
                                /* otherwise the reason the */
                                /* printer has been disabled. */
    short   p_flags;            /* See below for flag values. */
};

/* Value interpretation for p_flags: */

#define     P_ENAB 1            /* printer enabled */
#define     P_AUTO 2            /* disable printer automatically */
#define     P_BUSY 4            /* printer now printing a request */
```

FILES

These subroutines are located in the **libdev** library (**/usr/lib/libdev**).

NAME

getpw – get name from UID

SYNOPSIS

```
int getpw (uid, buf)
int uid;
char *buf;
```

DESCRIPTION

Getpw searches the password file for a user ID number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *Getpw* returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )
```

DESCRIPTION

Getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the **<pwd.h>** header file:

```
struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};

struct comment {
        char    *c_dept;
        char    *c_name;
        char    *c_acct;
        char    *c_bin;
};
```

This structure is declared in **<pwd.h>** so it is not necessary to redeclare it.

The *pw_comment* field is unused; the others have meanings described in *passwd*(**4**).

Getpwent when first called returns a pointer to the first **passwd** structure in the file; thereafter, it returns a pointer to the next **passwd** structure in the file; so successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user ID matching *uid* is found and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the

GETPWENT (3C)

particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

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

GETS (3S)

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until $n-1$ characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;
```

DESCRIPTION

Getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char    ut_user[8];         /* User login name */
        char    ut_id[4];           /* /etc/inittab id (usually line #) */
        char    ut_line[12];        /* device name (console, lnxx) */
        short   ut_pid;             /* process id */
        short   ut_type;            /* type of entry */
        struct  exit_status {
          short     e_termination;  /* Process termination status */
          short     e_exit;         /* Process exit status */
        } ut_exit;                  /* The exit status of a process
                                     * marked as DEAD_PROCESS. */
        time_t  ut_time;            /* time entry was made */
};
```

Getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

Getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

Getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the

line->ut_line string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file doesn't exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* if it finds that it isn't already at the correct place in the file will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the **<search.h>** header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table, and may be followed by another call to *hcreate*.

NOTES

Hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a manner similar to *strcmp* (see *string*(3C)).

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available.

START	Place new entries at the beginning of the linked list (default is at the end).
SORTUP	Keep the linked list sorted by key in ascending order.
SORTDOWN	Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining debugging printout (**–DDEBUG**) and for including a test driver in the calling routine (**–DDRIVER**). The source code should be consulted for further details.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

BUGS

Only one hash search table may be active at any given time.

NAME

hypot – Euclidean distance function

SYNOPSIS

```
#include <math.h>

double hypot (x, y)
double x, y;
```

DESCRIPTION

Hypot returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

L3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

 LDAHREAD (3X)

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If **TYPE(***ldptr***)** is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

Ldahread returns **SUCCESS** or **FAILURE**. *Ldahread* will fail if **TYPE(***ldptr***)** does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldopen(3X) and *ldclose* are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE(***ldptr***)** does not represent an archive file, *ldclose* will close the file and free the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE(***ldptr***)** is the magic number of an archive file, and if there are any more files in the archive, *ldclose* will reinitialize **OFFSET(***ldptr***)** to the file address of the next archive member and return **FAILURE**. The **LDFILE** structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

Ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE***(ldptr)*. *Ldaclose* always returns **SUCCESS**. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

Ldfhread returns **SUCCESS** or **FAILURE**. *Ldfhread* will fail if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro **HEADER(***ldptr***)** defined in **ldfcn.h** (see *ldfcn*(4)). The information in any field, *fieldname*, of the file header may be accessed using **HEADER(***ldptr***).***fieldname*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread(ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO linent;

int ldlinit(ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO linent;
```

DESCRIPTION

Ldlread searches the line number entries of the common object file currently associated with *ldptr*. *Ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *Ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlread, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE**. *Ldlread* will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *Ldlinit* will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *Ldlitem* will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek,ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnlseek seeks to the line number entries of the section specified by *sectname*.

Ldlseek and *ldnlseek* return **SUCCESS** or **FAILURE**. *Ldlseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnlseek* will fail if there is no section name corresponding with **sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

Ldohseek returns **SUCCESS** or **FAILURE**. *Ldohseek* will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen – open a common object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

Ldopen and *ldclose*(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL**, then *ldopen* will open *filename* and allocate and initialize the **LDFILE** structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE**(*ldptr*) is the archive magic number, *ldopen* will reinitialize the **LDFILE** structure for the next archive member of *filename*.

Ldopen and *ldclose* are designed to work in concert. *Ldclose* will return **FAILURE** only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*.

```
/* for each filename to be processed */
ldptr = NULL;
do
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, *ldaopen* will open *filename* anew and allocate and initialize a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. *Ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For

example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnrseek seeks to the relocation entries of the section specified by *sectname*.

Ldrseek and *ldnrseek* return **SUCCESS** or **FAILURE**. *Ldrseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnrseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char sectname;
SCNHDR *secthead;
```

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

Ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

Ldshread and *ldnshread* return SUCCESS or FAILURE. *Ldshread* will fail if *sectindx* is greater than the number of sections in the object file; *ldnshread* will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnsseek seeks to the section specified by *sectname*.

Ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. *Ldsseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnsseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it will return the the index of the next entry.

Ldtbindex will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

LDTBREAD (3X)

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

Ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

Ldtbread returns **SUCCESS** or **FAILURE**. *Ldtbread* will fail if *symindex* is greater than the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbseek – seek to the symbol table of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek return **SUCCESS** or **FAILURE**. *Ldtbseek* will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;
```

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

Fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order in order to establish lock with this function call.

Function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define F_ULOCK 0   /* Unlock a previously locked section */
#define F_LOCK  1   /* Lock a section for exclusive use */
#define F_TLOCK 2   /* Test and lock a section for exclusive use */
#define F_TEST  3   /* Test section for other process' locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

Size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

LOCKF (3C)

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a **−1** and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EBADF]
: *Fildes* is not a valid open descriptor.

[EACCES]
: *Cmd* is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK]
: *Cmd* is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), read(2), write(2)

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

WARNINGS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O pacakage is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

FILES

/etc/profile

SEE ALSO

env(1), login(1M), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch – linear search and update

SYNOPSIS

```
#include <stdio.h>
#include <search.h>

char *lsearch ((char *)key, (char *)base,
nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base,
nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *Key* points to the datum to be sought in the table. *Base* points to the first element in the table. *Nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *Compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

Lfind is the same as *Lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 30
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE],
        *lsearch( );
```

LSEARCH (3C)

```
unsigned nel = 0;
int strcmp( );
...
while (fgets(line, ELSIZE, stdin) != NULL &&
        nel < TABSIZE)
        (void) lsearch(line,(char *)tab,&nel,
                ELSIZE, strcmp);
```

SEE ALSO

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;
```

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Mallopt provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24.

M_NLBLKS — Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN — Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

M_KEEP — Preserve data in a freed block until the next *malloc, realloc,* or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the <*malloc.h*> header file.

Mallopt may be called repeatedly, but may not be called after the first small block is allocated.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk (2).

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation, or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

NOTE

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

NAME

matherr – error-handling function

SYNOPSIS

```
#include <math.h>

int matherr (x)
struct exception *x;
```

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named *matherr* in their programs. *Matherr* must be of the form described above. A pointer to the exception structure *x* will be passed to the user-supplied *matherr* function when an error occurs. This structure, which is defined in the **<math.h>** header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	domain error
SING	singularity
OVERFLOW	overflow
UNDERFLOW	underflow
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that had the error. The variables *arg1* and *arg2* are the arguments to the function that had the error. *Retval* is a double that is returned by the function having the error. If it supplies a return value, the user's *matherr* must return non-zero. If the default error value is to be returned, the user's *matherr* must return 0.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to non-zero and the program continues.

EXAMPLE

```
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
        case SING: /* print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
```

```
        case OVERFLOW:
                if (!strcmp("exp", x->name)) {
                        /* if exp, print message, return the argument */
                        fprintf(stderr, "exp of %f\n", x->arg1);
                        x->retval = x->arg1;
                } else if (!strcmp("sinh", x->name)) {
                        /* if sinh, set errno, return 0 */
                        errno = ERANGE;
                        x->retval = 0;
                } else
                        /* otherwise, return HUGE */
                        x->retval = HUGE;
                break;
        case UNDERFLOW:
                return (0); /* execute default procedure */
        case TLOSS:
        case PLOSS:
                /* print message and return 0 */
                fprintf(stderr, "loss of significance in %s\n", x->name);
                x->retval = 0;
                break;
        }
        return (1);
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL: y0, y1, yn (neg. no.)	– M, –H	– –	H –	0 –	– –	* –
EXP:	–	–	H	0	–	
POW: (neg.)**(non-int.), 0**0	– M, 0	– –	H –	0 –	– –	– –
LOG: log(0): log(neg.):	 – M, –H	 M, –H –	 – –	 – –	 – –	 – –
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	–	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, COSH:	–	–	H	–	–	–
SIN, COS:	–	–	–	–	M, 0	M, *
TAN:	–	–	H	–	0	*
ACOS, ASIN:	M, 0	–	–	–	–	–

ABBREVIATIONS	
*	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
-H	-HUGE is returned.
0	0 is returned.

NAME

memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

Memchr returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

Memcmp compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

Memcpy copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

Memset sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

NOTE

For user convenience, all these functions are declared in the optional **<memory.h>** header file.

BUGS

Memcmp uses native character comparison, which is signed on PDP-11s, unsigned on other machines.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

menu – display and accept menus

SYNOPSIS

```
#include <menu.h>
int menu(menu, op)
menu_t *menu;
int op;
```

DESCRIPTION

This routine manipulates a menu as determined by the operation code (*op*). If the *op* arg is M_BEGIN, the menu is initialized and displayed. If *op* is M_INPUT, user input is accepted. If *op* is M_END, the menu is terminated and removed from the display. If *op* is M_DESEL, all currently-selected items are de-selected before the menu is processed. These functions may be combined in many ways. By specifying (M_DESEL | M_BEGIN | M_INPUT | M_END), the caller creates a "pop-up" menu which is initialized (displayed), used for input, then removed. Generally, (M_BEGIN | M_INPUT) is used for the first call, M_INPUT for each subsequent interaction, and M_END when the menu is to be discarded. M_DESEL can be added to clear bad choices after an error.

During the M_INPUT function, the user may point to items with the mouse or with the keyboard (arrows, Prev, Next, Beg, Home, End). The user may mark entries with the mouse or keyboard (Slect, Mark), or may qualify one or more items by typing. The Cancl, Clear Line, Back Space, and Return keys perform their generic editing functions. For larger menus, the Roll and Page keys may be used to scroll through choices which do not fit in the prevailing window.

All other keys, including Enter, are returned to the caller.

The menu structure has the following form:

```
typedef struct
{
    char        *m_label;              /* menu label */
    char        *m_title;              /* menu title */
    char        *m_prompt;             /* menu prompt */
    char        m_rows;                /* desired rows */
    char        m_cols;                /* desired cols */
    char        m_iwidth;              /* item width */
    char        m_iheight;             /* item height */
    char        m_flags;               /* flags */
    char        m_lbuf[M_MAXLINE];     /* input buffer */
    int         m_win;                 /* window pointer */
    track_t     *m_track;              /* menu track list ptr */
    int         m_oldwidth;            /* last window width */
    int         m_oldheight;           /* last window height */
    int         m_selcnt;              /* count of # selected */
    mitem_t     *m_items;              /* pointer to items */
    mitem_t     *m_curi;               /* current item */
```

```
    mitem_t  *m_topi;                    /* item at top of dpy */
} menu_t;
```

M_label is displayed in the menu's window label line. If *m_label* is NULL, no label is displayed.

M_title is a title for the menu. If *m_title* is NULL, the menu is displayed without any title. If *m_title* contains newline characters, all but the first line of the title are underlined.

M_prompt (when non-NULL) is an prompt for the menu. All prompts are displayed on the system prompt line, generally located at the bottom of the display.

M_rows and *m_cols* allow the caller to specify the number of rows and columns of items. Either or both of these values may be zero, in which case *menu* will try to pick "good" values. It is quite common to specify "zero" rows and one column to force output to be vertical. When both *m_rows* and *m_cols* are zero, *menu* tries to fit the menu into an appealing rectangle.

M_iwidth specifies a maximum width for each item. If *m_iwidth* is zero, *menu* will display as much of the item as possible given the available window real-estate. If *m_iwidth* is non-zero, items are automatically truncated to that width, regardless of other parameters. This is very useful when generating multi-column menus where the caller wishes to prevent one long item from disrupting the columnar output. M_IWIDTH IS NOT YET IMPLEMENTED.

M_iheight allows the caller to specify the number of rows per item on the display. Newline characters in the item's name cause *menu* to advance to the next row. If *m_iheight* is zero, one-row items are assumed. M_IHEIGHT IS NOT YET IMPLEMENTED.

M_flags contains the M_SINGLE flag which prohibits the user from selecting more than one item from the menu. If the M_SINGLE flag is off, the user may select multiple items from the menu. The M_USEWIN flag, if set, will cause *menu* to use the window supplied in *m_win* instead of creating its own. If M_USEWIN is set, M_BEGIN operations will re-size the window (as necessary) and M_END operations do not delete the window. The M_WINNEW flag causes *menu* to use the "new" algorithm to place the window. Basically, the new algorithm looks for relatively empty screen space to place the window. M_WINSON causes *menu* to use the "son" algorithm which causes the new window to slightly overlap the current window. If neither M_WINNEW nor M_WINSON is given, the "popup" algorithm is used. This causes the new window to appear near the middle of the current window, inside it if possible. The M_NOMOVE, M_NOHELP, and M_NORESIZE flags, if set, prevent the Move, Help and Resize icons, respectively, from being displayed on the menu border. The M_ASISTITLE flag, if set, causes the menu title to be displayed as supplied by the user, i.e., without centering.

M_lbuf is an array of characters which is used to assemble typed input. It is always returned to the caller in case it is necessary to record keystrokes. In addition, the user may enter keystroke data which does not match any items. In this case *m_selcnt* is set to zero on return.

M_win holds the window identifier associated with this menu. It is allocated on an M_BEGIN call, used on subsequent calls, and deleted on an M_END call. *M_track* is a pointer to the mouse-tracking information required during menu interaction. The space for this data is allocated on M_BEGIN and freed on M_END.

The caller should not use *m_track*, *m_oldwidth*, or *m_oldheight*. They may be left un-initialized on call to M_BEGIN. If a value must be given (as in a *mitem_t* initializer), a value of zero should be used.

On return, *m_selcnt* contains a count of the number of selected items.

M_items points to the array of menu items (see below). *M_curi* points to the current item. The caller should point *m_curi* to the default item. *Menu* will modify *m_curi* as the user moves the highlighting around in the menu. The list of menu items is terminated by an item whose *mi_name* is NULL. *M_topi* contains a pointer to the item which is at the top of the display. As the window scrolls through the menu, *m_topi* changes to reflect the new position of the view. The caller needn't initialize *m_topi* as *menu* will compute a "good" initial view from the value in *m_curi*. In general, callers will not read the value in *m_topi* and should never write it.

Each item in the array pointed to by *m_items* and *m_curi* has the following form:

```
typedef struct
{
    char        *mi_name;   /* name of item        */
    char        mi_flags;   /* flags            */
    int         mi_val;     /* user-supplied value */
} mitem_t;
```

Mi_name is the item name, *mi_flags* contains the M_MARKED flag which indicates that this item should be marked (on call) and/or was marked (on return) and/or the M_DIMMED flag which indicates that this item should be displayed in lower-intensity. *Mi_val* is unused by *menu* and is available for application-specific data. Often, *mi_val* contains a small positive integer identifying the particular choice.

EXAMPLE

The following program illustrates a simple single-selection menu:

```
#include <tam.h>
#include <menu.h>
#include <stdio.h>
#include <kcodes.h>
```

```
mitem_t cmditems[] =
{
        "Add",          0,0,
        "Advance",      0,1,
        "Backspace",    0,2,
        "Copy",         0,3,
        "Create",       0,4,
        "Delete",       0,5,
        "Dump",         0,6,
        "Examine",      0,7,
        "Exit",         0,8,
        "Finish",       0,9,
        " *Weird",      0,10,
        0,              0,0
};

menu_t cmdmenu =
{
        "Single",
        "Commands",
        "Pick a command from the list",
        0,1,0,0,
        M_SINGLE,
        {0},
        0,0,0,0,0,
        cmditems,
        cmditems,
        0
};

mitem_t multitems[ ] =
{
        "January",      0,1,
        "February",     0,2,
        "March",        0,3,
        "April",        0,4,
        "May",          0,5,
        "June",         0,6,
        "July",         0,7,
        "August",       0,8,
        "September",    0,9,
        "October",      0,10,
        "November",     0,11,
        "December",     0,12,
        "Monday",       0,13,
        "Tuesday",      0,14,
        "Wednesday",    0,15,
        "Thursday",     0,16,
        "Friday",       0,17,
        "Saturday",     0,18,
        "Sunday",       0,19,
        0,              0,0
};
```

```
menu_t multmenu =
{
        "Multiple",
        "Many Choices!",
        "Pick multiple commands from the list",
        0,0,0,0,
        M_WINSON,
        {0},
        0,0,0,0,0,
        multitems,
        multitems,
        0
};



main()
{
        int err;
        int cmdop,multop;
        char *which;

        winit();
        keypad(0,1);

        cmdop = M_BEGIN | M_INPUT;
        multop = M_BEGIN | M_INPUT;

        while(1)
        {
                which = "cmdmenu";
                err = menu( &cmdmenu, cmdop );
                cmdop &= ~M_BEGIN;
                if ( err < 0 || err == Close )
                        break;

                which = "multmenu";
                err = menu( &multmenu, multop );
                multop &= ~M_BEGIN;
                if ( err < 0 || err == Close )
                        break;
        }
        if ( err < 0 )
        {
                fprintf(stderr,"err %d in %s",err,which);
                sleep(5);
        }
        wexit(0);
}
```

FILES

/usr/include/menu.h
/usr/include/kcodes.h

SEE ALSO

form(3T), tam(3T).

DIAGNOSTICS

Menu returns non-negative keyboard codes (see **kcodes.h**) when keyboard input terminated the menu interaction. Other return values signal more serious errors and are defined in **menu.h**.

NAME

message – display error and help messages

SYNOPSIS

```
#include <message.h>

int message(mtype, hfile, htitle, format [, arg ] ... )
int mtype;
char *hfile, *htitle, *format;

int exhelp(hfile, htitle)
char *hfile, *htitle;
```

DESCRIPTION

Message formats the passed message a la *printf* and displays the message in a window that *message* creates. The message is automatically wrapped to fit within the dimensions of the window, and may contain embedded newlines. *Message* then waits for user input and returns the character read to the caller.

Mtype can have one of the following values:

MT_HELP	Displays help message
MT_ERROR	Displays error message
MT_POPUP	Displays a popup window
MT_QUIT	Displays error message with cancel option
MT_CONFIRM	Displays confirmation message
MT_INFO	Displays informational message

All message types except MT_POPUP display the available choices (ENTER, CANCL, or HELP) and beep any other keys. The MT_INFO message type takes the first line of the message and uses it as the window label.

When HELP is selected, message executes *uahelp*, passing it *hfile* and *htitle* as the help file name and initial help display title. *Message* then waits for *uahelp* to return. If *hfile* is NULL, then the HELP choice is not offered or accepted.

Exhelp executes *uahelp* directly, without going through an intermediate help display. In both cases, if *hfile* is a full path name then it is passed to *uahelp* as is, otherwise the pathname */usr/lib/ua* is assumed.

EXAMPLES

To print an error message when a file isn't found:

```
message(MT_ERROR, "ua.hlp", "System errors",
                                "%s not found", name);
```

If the user presses the Help key in response to this message, then *uahelp* will display the page on system errors in the user agent help file.

To get confirmation:

```
message(MT_CONFIRM, NULL, NULL,
                    "%s will be overwritten",
                    name);
```

In this case, if the user presses Help, the key will be declared invalid.

DIAGNOSTICS

Message returns the character typed by the user, or −1 if error. Currently, the only error is *Can't create window*, and in this case, *message* will display this error message on the prompt line of the current window.

Exhelp returns −1 on error (argument error, fork failure, or exec failure).

SEE ALSO

uahelp(1), tam(3T).

MKTEMP (3C)

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing Xs; *mktemp* will replace the Xs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

NAME

monitor – prepare execution profile

SYNOPSIS

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short *buffer;
int bufsize, nfunc;

DESCRIPTION

An executable program created by **cc −p** automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **−p** of *cc*(1) are recorded. (The C Library and Math Library supplied when **cc -p** is used also have call counts recorded.) For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor ((int (*)())2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor ((int (*)())NULL, 0, 0, 0, 0);
```

Prof(1) can then be used to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

NLIST (3C)

NAME

nlist – get entries from name list

SYNOPSIS

```
#include <a.out.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl[ ];
```

DESCRIPTION

Nlist examines the name list in the executable file whose name is pointed to by *file-name*, and selectively extracts a list of values and puts them in the array of nlist structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(4) for a discussion of the symbol table structure.

This subroutine is useful for examining the system name list kept in the file **/unix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(4).

DIAGNOSTICS

All type entries are set to 0 if the file cannot be read or if it doesn't contain a valid name list.

Nlist returns –1 upon error; otherwise it returns 0.

NAME

paste – paste buffer utilities

SYNOPSIS

```
#include <pbf.h>

FILE *pb_open()

int pb_check(stream)
FILE *stream;

int pb_seek(stream)
FILE *stream;

int pb_empty(stream)
FILE *stream;

char *pb_name()

int pb_puts(ptr,stream)
char *ptr;
FILE *stream;

int pb_weof(stream)
FILE *stream;

char *pb_gets(ptr, n, stream)
char *ptr;
int n;
FILE *stream;

int pb_gbuf(ptr, n, fn, stream)
char *ptr;
int n;
int (*fn) ();
FILE *stream;

char *adf_gtwrd(sptr, dptr)
char *sptr, *dptr;

char *adf_gtxcd(sptr, dptr)
char *sptr, *dptr;

int adf_gttok(ptr, tbl)
char *ptr;
struct s_kwtbl *tbl;
```

DESCRIPTION

Pb_open opens the paste buffer file and associates a *stream* with it. *Pb_open* returns a pointer to the **FILE** structure associated with the stream.

Pb_check determines if the paste buffer file associated with *stream* contains any data, and returns **TRUE** if the paste buffer is not empty, and **FALSE** otherwise.

Pb_seek scans the paste buffer file from the beginning for the end of file, and then seeks to that displacement. It sets up the paste buffer for appending.

Pb_empty empties the paste buffer file and closes it. It is intended to be called after the paste buffer file has been read.

Pb_name returns a pointer to a static area containing the name of the paste buffer file.

Pb_puts appends a null terminated text string to the paste buffer file in ADF format. It has the same interface as *fputs*.

Pb_weof writes an end of file code to the paste buffer file, and closes the file. It is intended to be called after the paste buffer file has been written.

Pb_gets reads the next string from the paste buffer file and converts it to text. It has the same interface as *fgets*. *Pb_gets* always returns EOF after 511 bytes have been read. To read larger blocks of text, *pb_gbuf* must be used.

Pb_gbuf reads a paste buffer file entry and converts it to text. It puts the results into the buffer passed to it. *Pb_gbuf* calls the passed function to store the buffer at the end of the paste entry, or when the buffer becomes full. When this function is called, it is passed the buffer address, and the number of bytes in the buffer. This function should return a negative value on error.

If a **NULL** pointer is passed in place of the function, then *pb_gbuf* returns a null-terminated string, stored in the passed buffer. Note that *pb_gbuf* can only be called once, and the paste buffer file should be marked empty after the call.

The three functions *adf_gtwrd*, *adf_gtxcd*, and *adf_gttok* are utilities used by *pb_gbuf* to interpret the ADF format. Applications that wish to access the ADF formatted files directly might use them.

adf_gtwrd scans the input string pointed to by *sptr* and pulls out the next word. It stores this word (null-terminated) to the buffer pointed to by *dptr*, and returns an updated input pointer.

Adf_gtxcd pulls out the embedded text code from the input string pointed to by *sptr*, and also returns an updated input pointer. This text code is stored null-terminated in the buffer pointed to by *dptr*. *Adf_gtxcd* should be called when a "\" is encountered in the input line in a text field, and *sptr* should be pointing to the character following the "\."

Adf_gttok converts a word (a null-terminated string) to a token (integer) and returns the token. The conversion is driven by the passed keyword table. The keyword table structure is defined in the include file **pbf.h**.

EXAMPLE

To implement the cut operation:

```
paste_file = pb_open ();
if (paste_file == NULL)
   message (MT_ERROR, "wp.hlp", "Paste",
         "Unable to open or create paste buffer file");
```

```
else
   {
   chr = Enter;
   if (pb_check (paste_file))
      {
      chr = message (MT_QUIT, "wp.hlp", "Paste",
            "Paste buffer contains an entry - \ do you
wish to overwrite?");
      }
   if (chr == Enter)
      {
         ... /* Paste file is empty, or is OK to */
         ... /* overwrite, write the paste file using */
         ... /* fwrite, fprintf, etc.
      }
   fclose (paste_file);
   }
```

To implement the paste operation:

```
paste_file = pb_open ();
if (paste_file == NULL)
   message (MT_ERROR, "wp.hlp", "Paste",
         "Unable to open or create paste buffer file");
else
   {
   if (pb_check (paste_file))
      {
      ... /* Paste file exists and is non empty, */
      ... /* Read the paste file using fread,    */
      ... /* fscanf, etc.
      }
   else
      message (MT_ERROR, "wp.hlp", "Paste",
            "Paste buffer is empty");
   pb_empty (paste_file);
   }
```

SEE ALSO

adf(4), message(3T), tam(3T).

DIAGNOSTICS

Pb_open returns a NULL pointer on failure. *Pb_gets* returns a NULL pointer at end of file. *Pb_puts* returns EOF on failure.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns –1 if *stream* is not associated with a "*popen* ed" command.

BUGS

If the original and "*popen* ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*; see *fclose*(3S).

PRINTF (3S)

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

#include <stdio.h>

int printf (format [**,** arg] . . . **)**
char *format;

int fprintf (stream, format [**,** arg] . . . **)**
FILE *stream;
char *format;

int sprintf (s, format [**,** arg] . . . **)**
char *s, format;

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (**\0**), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width;

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string: a null digit string is treated as zero.

An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or −).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (**X**) conversion, a non-zero result will have **0x** (**0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[−]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double *arg* is converted in the style "[−]d.ddde±dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point

appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than −4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer *arg* has the value zero, the result is undefined. A *null* arg will yield undefined results.

% Print a **%**; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d",
weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

int putchar (c)
char c;

int fputc (c, stream)
char c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *Putchar*(*c*) is defined as *putc*(*c, stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw writes the word (i.e. integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen*(see *fopen*(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered information is queued for writing on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block; when it is line-buffered each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *Setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant **EOF**. This will occur if

the file *stream* is not open for writing, or if the output file cannot be grown. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor. For this reason the use of *putw* should be avoided.

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

String points to a string of the form "*name=value.*" *Putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

DIAGNOSTICS

Putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

WARNINGS

Putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.

This routine uses *malloc*(3C) to enlarge the environment.

After *putenv* is called, environmental variables are not in alphabetical order.

A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwuid* writes a line on the stream *f* which matches the format of **/etc/passwd**.

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

PUTS (3S)

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout*.

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES

Puts appends a new-line character while *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned int nel;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

 REGCMP (3X)

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

char *regcmp(string1 [, string2, . . .], 0)
char *string1, *string2, . . .;

char *regex(re, subject[, ret0, . . .])
char *re, *subject, *ret0, . . .;

extern char *loc1;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * .^ These symbols retain their current meaning.

$ Matches the end of the string, **\n** matches the new-line.

– Within brackets the minus means *through*. For example, **[a–z]** is equivalent to **[abcd . . .xyz]**. The – can appear as itself only if used as the last or first character. For example, the character class expression **[]–]** matches the characters] and –.

+ A regular expression followed by + means *one or more times*. For example, **[0–9]+** is equivalent to **[0–9][0–9]***.

{m} {m,} {m,u}
Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(. . .)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At present, at most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, e.g. *, +, { }, can work on a single character or a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        . . .
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
        . . .
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        . . .
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) reuses the same vector saving time and space:

```
/* user's program */
        . . .
malloc(n) {
        static int rebuf[256];
        return rebuf;
}
```

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf (format [ , pointer ] ... )
char *format;

int fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;

int sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optionally signed integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset,* and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex, (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus [0123456789] may be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**

, f , and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
scanf ("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf(3S), strtod(3C), strtol(3C).

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

SETBUF (3S)

NAME

setbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is a NULL character pointer input/output will be completely unbuffered.

A constant **BUFSIZ,** defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

Setbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in **stdio.h**) are:

_IOFBF causes input to be fully buffered.

_IOLBF causes output to be line buffered; the buffer will be flushed when a new line is written, the buffer is full, or input is requested.

_IONBF causes input and output to be completely unbuffered.

A buffer is normally obtained from *malloc*(3C) at the time of the first *getc* or *putc*(3S) on the file, except that the standard error stream *stderr* is normally not buffered.

Output streams directed to terminals are always line-buffered unless they are unbuffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S).

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the **<setjmp.h>** header file), for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2).

WARNING

If *longjmp* is called when *env* was never primed by a call to *setjmp*, or when the last such call is in a function which has since returned, absolute chaos is guaranteed.

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

Sinh, *cosh* and *tanh* return respectively the hyperbolic sine, cosine and tangent of their argument.

DIAGNOSTICS

Sinh and *cosh* return **HUGE** when the correct value would overflow, and set *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*; if the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the *sleep* routine returns, but if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sputl, sgetl – access long numeric data in a machine independent fashion

SYNOPSIS

```
sputl ( value, buffer )
long value;
char *buffer;

long sgetl ( buffer )
char *buffer;
```

DESCRIPTION

Sputl(3X) will take the 4 bytes of the long *value* and place them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines. *Sgetl* will retrieve the 4 bytes in memory starting at the address pointed to by *buffer* and return the long value in the byte ordering of the host machine.

The usage of *sputl*(3X) and *sgetl* in combination provides a machine independent way of storing long numeric data in an ASCII file. The numeric data stored in the portable archive file format (see *ar*(4)) is written and read into/from buffers with *sputl*(3X) and *sgetl* respectively.

A program which uses these functions must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ar(4).

NAME

ssignal, gsignal – software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

Gsignal raises the signal identified by its argument, *sig*:

If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.

If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.

If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.

If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

SEE ALSO

signal(2), kill(2).

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file.

A constant **NULL** (0) designates a nonexistent pointer.

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

An integer constant **EOF** (−1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdipc – standard interprocess communication package

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2) and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget* and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

Ftok returns **(key_t) −1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments *s1*, *s2* and *s* point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy* and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* appends at most *n* characters. Each returns a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*. *Strncmp* makes the same comparison but looks at *n* characters at most.

Strcpy copies string *s2* to *s1*, stopping after the null character has been copied. *Strncpy* copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

Strlen returns the number of characters in *s*, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

NOTE

For user convenience, all these functions are declared in the optional **<string.h>** header file.

BUGS

Strcmp and *strncmp* use native character comparison, which is signed on PDP-11s, unsigned on other machines.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

DESCRIPTION

Strtod returns as a double-precision floating-point number, the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

Strtod recognizes an optional string of "white-space" characters [as defined by *isspace* in *ctype*(3C)], then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

Atof(str) is equivalent to *strtod(str (char **)NULL)*.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C).

DIAGNOSTICS

If the correct value would cause overflow, =HUGE (as defined in **<math.h>**) is returned (according to the sign of the value), and *errno* is set to ERANGE. If the correct value would cause underflow, zero is returned and *errno* is set to ERANGE.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

DESCRIPTION

Strtol returns as a long integer the value represented by the character string *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in **ptr*. If no integer can be formed, **ptr* is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thus: After an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment, or by an explicit cast.

Atol(str) is equivalent to *strtol(str, (char **)NULL, 10).*

Atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10).*

SEE ALSO

scanf(3S), strtod(3C).

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*−1 instead. If *nbytes* is negative *swab* does nothing.

NAME

system – issue a shell command

SYNOPSIS

```
#include <stdio.h>

int system (string)
char *string;
```

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System forks to create a child process that in turn exec's */bin/sh* in order to execute *string*. If the fork or exec fails, *system* returns −1 and sets *errno*.

NAME

tam – a library of calls that supports terminal access, including windows.

SYNOPSIS

```
#include <tam.h>

winit()

wexit(rval)

int iswind()

int wcreate(row,col,height,width,flags)
short row,col,height,width;
unsigned short flags;

int wdelete(wn)
short wn;

int wselect(wn)
short wn;

int wgetsel()

int wgetstat(wn,wstatp)
short wn;
WSTAT *wstatp;

int wsetstat(wn,wstatp)
short wn;
WSTAT *wstatp;

int wputc(wn,c)
short wn;
char c;

int wputs(wn,cp)
short wn;
char *cp;

int wprintf(wn,fmt,arg1 ... argn)
short wn;
char *fmt;

int wslk(wn,kn,llabel,slabel)
short wn;
short kn;
char *llabel, *slabel;

int wslk(wn,0,slong1,slong2,sshort) /* alternate form of
wslk */
short wn;

char *slong1,*slong2,*sshort;

int wcmd(wn,cp)
short wn;
char *cp;

int wprompt(wn,cp)
short wn;
char *cp;
```

```
int wlabel(wn,cp)
short wn;
char *cp;

int wrefresh(wn)
short wn;

int wuser(wn,cp)
short wn;
char *cp;

int wgoto(wn,row,col)
short wn,rol,col;

int wgetpos(wn,rowp,colp)
short wn;
int *rowp,*colp;

int wgetc(wn);
short wn;

char *kcodemap(code);
unsigned char code;

int keypad(dummy,flag)
int dummy,flag;

int wsetmouse(wn,ms);
short wn;
struct umdata *ms;

int wreadmouse(wn,xp,yp,bp,rp)
short wn;
int *xp,*yp,*bp,*rp;

int wprexec()

int wpostwait()

wnl (wn,flag)
short wn;
int flag;

wicon(wn, row, col, icp)
short wn, row, col;
struct icon *icp;

wicoff(wn, row, col, icp)
short wn, row, col;
struct icon *icp;

wtargeton()
```

DESCRIPTION

The Terminal Access Method (TAM) routines provide a device-independent ANSI X3.64 interface to terminals. TAM also provides calls for creating, manipulating, and displaying to windows and can support an optional mouse input device.

Multiple overlapping windows can appear simultaneously on the screen. The characteristics of a window are its dimensions (height and width), position on the screen, and position relative other simultaneous windows. The UNIX PC kernel orders windows

according to which window is obscuring which other windows. The top window in the sequence is always completely visible. When a window is selected, it moves to the top, potentially overlaying windows that were previously visible. For windows with borders, row and column indicate the upper-left corner of the *outside* of the window, while height and width describe the dimensions of the *inside* of the window. Of the flags, only NBORDER applies to remote terminal windows. These flags are defined in *window*(7) in **sys/window.h**; note that **tam.h** calls both **window.h** and stdio.h .

The *iswind()* call determines if the terminal is local or remote. The differences in the way TAM provides windows for bit-mapped terminals versus remote terminals are described below.

For the UNIX PC bit-mapped screen terminal, either the same process (application) or distinct processes can own simultaneous windows. The UNIX PC kernel remembers the covered portions of any window on a pixel basis. Each window is a separate device and can have its *tty* modes set independently of the other windows.

Since file descriptors are used to access windows, a child process can inherit the open windows of its parent. All of the usual UNIX system calls apply, so an application can set the "close on exec" bit via *fcntl()* to prevent access to particular windows by a child process.

A process can also set up a child process so that its standard input and output point to a particular window by the usual technique-closing file descriptors 0 and 1 (*stdin* and *stdout*) and duplicating the window's file descriptor. Window changed signals are sent to the process group associated with the particular window. Thus, by default, a parent process receives signals concerning windows created by its children. To prevent a parent from receiving these signals, use the *setpgrp()* system call.

In the case when the child process is unaware of windows, the following parameters should also be fixed: *tty* modes should be set to reasonable values and the signal SIGWIND should be ignored rather than caught. (SIGWIND is ignored by default.)

A window which is owned by a particular application can be manipulated without the consent of the application. A signal (SIGWIND) is sent to the process group associated with the window to report any window changes. If a single process creates multiple windows, it receives the SIGWIND signal when any of its windows are changed. To respond appropriately to changes in its window, an application must catch SIGWIND (the default is to ignore SIGWIND) and issue a *wgetstat()* (described below) to determine which windows changed and in what ways.

If an application changes characteristics of its own window, no signal is sent to the application. In fact, when any process in the process group associated with the window changes the window characteristics, no signal is sent. Some care should be taken when an application changes its own window, since this might defeat the wishes of the operator. For example, if a process makes its

window as large as the screen and every time it receives the SIGWIND signal it resets its window to the top and as large as the screen, the user is effectively blocked from running any other applications.

The major difference between the remote and UNIX PC terminals from the application point of view is the lack of kernel level window support for remote terminals. TAM supports windows transparently within a given process, but knows nothing about windows created by other processes. Remote terminals can display multiple windows but, unlike the UNIX PC terminal, only one application owns the screen at a time. TAM simulates the kernel window functions for a particular application by maintaining a screen image, as well as those portions of windows that are overlayed. Each process controls the entire screen, and windows created by other processes are erased when the new process issues a *winit()* call.

TAM uses the insert and delete line functions on remote terminals for scrolling whenever the window is as wide as the screen. This applies to windows with or without borders. Application programmers who are concerned with making their programs run efficiently on remote terminals should use windows of full screen width if planning on scrolling the window.

If an application creates a child process that writes to the screen, then after the child process dies, the application must issue a special *wrefresh()* call to restore its screen image. For remote terminals, TAM provides window ID's, instead of file descriptors, for accessing windows. Other differences from the bit-mapped terminal are that the *tty* modes cannot be independently set and child processes do not inherit windows.

TAM supports an optional mouse input device. Initially, mouse reports are disabled, so applications that don't use the mouse don't need to worry about it. When mouse reports are turned on by the application, they are returned to the application as a special 8-bit input code (or 7-bit input escape sequence) in the input stream.

In the default enabled state, a mouse report is inserted in the input stream on each change of the button state (whether each of the three buttons is up or down). Thus, these reports are buffered and are in sequence with any keyboard input.

The mouse report contains the mouse button state and the mouse cursor coordinates. This mouse report can be read and parsed by the application, or the application can use the *wgetmouse()* call, which reads the information from the input stream and returns it in a structure.

After being enabled, the mouse reports initially return only changes in mouse buttons. Optionally, mouse reports can tell if the mouse cursor goes outside (or inside) a specified rectangle. In the case of the remote terminal, mouse reports are never received, and the mouse related subroutine calls are ignored.

TAM routines are described below.

winit() The *winit()* call sets up the process for window access. *Winit()* must be called before any of the other window calls.

wexit() This should be called in place of *exit()*. *Wexit()* is the same as *exit()* but also resets the parameters set by *winit()* (e.g., *tty* modes).

iswind() Determines if the terminal is local or remote. *iswind()* is boolean–if true, the screen is bit-mapped.

wcreate() Creates a window. Arguments are the row, column of the top left corner of the window, the height and width of the window, and flags. The flags include whether or not the window has a border and whether or not variable character widths are allowed. The flags are described in **sys/window.h**. *wcreate()* returns a window number, *wn*, used in subsequent calls to that window. If *wcreate()* fails (returns –1), the programmer should direct an error message to the previous window.

wdelete() Deletes a specified window (*wn*). If the deleted window is on top and other windows are below it, the previously obscured windows become visible.

wselect() Selects the specified window (*wn*) as the current or active one. If the window is covered, it moves to the top. A window is implicitly selected when it is created (*wcreate*) or modified (*wsetstat*).

wgetsel() Returns the *wn* of the currently selected window.

wgetstat() Returns the information in WSTAT for a specified window (*wn*). Arguments are *wn* and the pointer to WSTAT. The content of WSTAT is:

```
struct wstat
{
        short           begy,begx,height,width;
        unsigned short  uflags; }
};
typedef struct wstat WSTAT;
```

The information includes the position and dimensions of the window, whether or not borders are on or off, and whether or not variable width characters are allowed.

wsetstat() Sets the status for a specified window (*wn*). *Wsetstat()* changes the parameters in WSTAT for a specified window and selects the window implicitly.

wputc() Outputs a specified character to a specified window (*wn*).

wputs() Outputs a specified character string to a specified window (*wn*). (Similar to *wputc()* above.)

wprintf() *wprintf()* does *printf()*'s to a specified window (*wn*). (Similar to *wputc()* above.)

In output to windows, a subset of the ANSI X3.64 escape sequences may be sent, and they are translated, if possible, for the particular terminal (see *termcap*(5) for information on defining to TAM the sequences recognized by a particular terminal). In the following four output routines, however, only standard ASCII text characters can be sent.

wslk() Outputs a null-terminated string to a screen labeled key, lines 28 and 29 on the screen. The arguments include the specified window (*wn*), the key number, and at least one character string. The first form of *wslk()* writes a single key; the alternate form writes all the screen labeled keys at once more efficiently. *kn*=0 indicates that this *wslk()* call is the alternate form. *Slong1* and *slong2* point to two 80-char strings of long SLK labels (16 characters each, 8 for the top label line followed by 8 for the bottom label line). *Sshort* points to a single 80-char string of short SLK labels (8 characters each).

wcmd() Outputs a null-terminated string to the command entry/echo line, line 27 on the screen. The arguments are the specified window (*wn*) and the character string.

wprompt() Outputs a null-terminated string to the prompt line, line 26 on the screen. The arguments are the specified window (*wn*) and the character string.

wlabel() Outputs a null-terminated string to the window label line in the top window border. The arguments are the specified window (*wn*) and the character string.

wrefresh() Flushes all output to the specified window. Output is normally buffered until input is read from the window. *Wrefresh*(=−1) refreshes the entire screen. If the terminal is remote, the *wrefresh()* call redisplays all windows known to the application in the remote terminal.

wuser() Writes the "user line" of the window. The user line is displayed by the *wmgr* process whenever it displays a list of windows.

wgoto() Moves the window's cursor to a specified row, column within the window. Arguments are *wn* and the row, column.

wgetpos() Gets the current position (row, column) of the cursor in the specified window (*wn*). Arguments are *wn* and the pointers to the row, column position of the cursor.

kcodemap() When passed an 8-bit value, *kcodemap()* returns a pointer to the 7-bit escape sequence that maps into that value.

wgetc() Gets a single character from the specified window (*wn*). *Wgetc()* is the window equivalent of *getchar()*. The input stream from any keyboard is translated into UNIX PC keyboard equivalents.

keypad() Determines how function keys are returned in a *wget()* call. There are three states:

flag=0 sets the 7-bit mode.
Function keys return escape sequences for a *wgetc()* call.

flag=1 sets the 8-bit mode.
Function keys return a single 8-bit character.

flag=2 sets the non-mapped mode.
Function keys return the code(s) generated by the terminal used.

wsetmouse() Sets up the parameters associated with the mouse. *Wsetmouse()* also takes a pointer to an *umdata* structure, which determines the report conditions for mouse motion and/or button changes. (See the discussion in the *window*(7) manual page about the umdata structure for specific usage.)

wreadmouse() Gets the mouse state, including the coordinates of the mouse cursor and whether each of the three mouse buttons is up or down. For a detailed description of this information, see *window*(7). The information is read from the input stream, so this routine should be called only after a mouse code is returned by *wgetc()*. The structure is defined in **sys/mouse.h**.

wprexec() This is called by the child process after a *fork()* and before an *exec()* to perform the appropriate actions for passing a window to a child process. "Appropriate action" varies from the bit map screen to the remote terminal. On the bit map, *wprexec()* creates a new window and passes it as *stdin*, *stdout*, and *stderr*. On remote terminals, *wprexec()* prepares the screen to be taken over by the child by flushing output and resetting *tty* modes.

wpostwait() This is called by the parent process after performing a *wait()* for a child process to reverse the effects of *wprexec*().

wnl(wn,flag) Turns on/off mapping of NL into CR/NL on output. The default stat is on.

wicon() Displays an icon (on the bit-mapped screen) at the specified row and column.

wicoff() Turns the icon off (blanks the area occupied by the icon).

wtargeton() Activates touch target capabilities for *menu, form,* and *message* on the 510a terminal. The default is off.

The command that compiles the code is

cc [flags] files -ltam -ltermcap [libraries]

For Curses compatibility calls see *#defines* in **tam.h** as well as the following:

initscr()
nl()
nonl()
cbreak()
nocbreak()
echo()
noecho()
getch()
flushinp()
attron()
attroff()
savetty()
resetty()
printw()
fixterm()
resetterm()

FILES

/usr/lib/ua/keynames
/usr/lib/ua/keymap
/usr/lib/ua/kmap.s4
/usr/lib/ua/tam.a

SEE ALSO

font(4), form(3T), menu(3T), window(7), kbd(7), escape(7), track(3T), wrastop(3T), shlib(4), message(3T).

DIAGNOSTICS

Unless otherwise specified, all functions return −1 on failure.

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ( )

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding FILE pointer. The file will automatically be deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S).

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

Tmpnam always generates a file name using the path-name defined as **P_tmpdir** in the **<stdio.h>** header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in **<stdio.h>**; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the path-name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a path-name for an appropriate directory, the path-name defined as **P_tmpdir** in the **<stdio.h>** header file is used. If that path-name is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is a path-name for the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

Tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc* failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen* or *creat* are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

track – track mouse motion

SYNOPSIS

```
#include <track.h>
int track(w, trk, op, butptr, whyptr)
int w, op, *butptr, *whyptr;
track_t *trk;
```

DESCRIPTION

Track allows the process to define an arbitrary number of "mouse motion rectangles" and to learn when the mouse ventures into any of them. In addition, *track* monitors keyboard input and mouse button presses, returning each as appropriate.

Each motion rectangle has an x,y address, a width and height, and an optional mouse icon to be displayed whenever the mouse is located in that rectangle.

The first argument is the window descriptor. *Trk* is a pointer to a track structure (see below). *Op* is T_BEGIN to initialize the track operation, T_INPUT to accept mouse and keyboard input, and T_END to terminate a track operation. These may be combined to produce complex operations: T_BEGIN | T_INPUT will initialize and begin tracking, etc. *Butptr* and *whyptr* are pointers to return values. The **int** pointed to by *butptr* will receive the 3-bit integer corresponding to the current mouse button state. *Whyptr* points to an **int** which will receive the "reason" for the return–one of MSIN, MSUP, or MSDOWN. MSIN means the mouse has moved to a new rectangle (or the background), MSUP and MSDOWN report mouse button changes.

The *track* structure controls track's operation:

```
typedef struct
{
        char            t_flags;        /* flags */
        char            t_scalex;       /* x & y scaling */
        char            t_scaley;
        short           t_lastx;        /* last known x,y pos */
        short           t_lasty;
        struct icon     *t_bicon;       /* background icon */
        struct umdata   t_umdata;       /* save the mouse data */
        tkitem_t        *t_tkitems;     /* ptr to items */
        tkitem_t        *t_curi;        /* ptr to current item */
} track_t;
```

T_flags contains either or both of MSUP and/or MSDOWN, which enable mouse button reporting. If both flags are zero, button presses have no effect–only motion rectangle transitions wake the process.

T_scalex and *t_scaley* determine the scaling factor for all the coordinates in the track items (see below). 1,1 gives unity scaling, allowing the application to specify pixel coordinates. Values of 0 for either scale parameter cause *track* to substitute the

appropriate character scale values. Thus, values of 0,0 for the scaling parameters specify that all user-supplied coordinates are in characters.

T_lastx and *t_lasty* are used internally by track to record the last known x,y position of the mouse. In particular, when *track* returns to the caller, *lastx* and *lasty* will contain the position which caused the return.

T_bicon is an optional icon to be used whenever the mouse is located in the background (not in any rectangle).

T_umdata is used internally by *track* to record the state of the mouse parameters on T_BEGIN and to restore them on T_END.

t_tkitems points to an array of track items (rectangles) which are described below. The list is terminated by a rectangle whose x, y, width, and height are all zero.

T_curi is a pointer to the current track item. On call, *track* assumes the mouse is located within the rectangle pointed to by *curi*. On return, *curi* points to the new current rectangle. A value of 0 means the background. *T_curi* is set to 0 on T_BEGIN.

Each track item (rectangle) has the following structure:

```
typedef struct
{
        unsigned short    ti_x;       /* x position */
        unsigned short    ti_y;       /* y position */
        unsigned short    ti_w;       /* width */
        unsigned short    ti_h;       /* height */
        struct icon       *ti_icon;   /* icon */
        int               ti_val;     /* user value */
} tkitem_t;
```

The first four parameters determine the location and size of the rectangle. 0,0 is the upper-left corner. Each of these parameters is scaled by the scaling factors.

Ti_icon points to an optional icon to be associated with this rectangle. Whenever the mouse is located within the rectangle, this particular icon is displayed.

Ti_val is a user-supplied value which is not used in the tracking process.

FILES

/usr/include/track.h
/usr/include/sys/window.h
/usr/include/kcodes.h

SEE ALSO

tam(3T), window(7).

DIAGNOSTICS

Track returns a key code (see **kcodes.h**) which determines what key was pressed. 'Mouse' is returned when a mouse event

occurred–the current item points to the track item in which the mouse is located (0 means the background). The button state and wakeup reason are also recorded. If a keyboard key is the cause of the return, these values are not necessarily updated.

Track can also return TERR_IOCTL if a system *ioctl* fails, or TERR_OK when no error occurred on a T_BEGIN or T_END operation where no input was performed.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

DESCRIPTION

Sin, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, which is in radians.

Asin returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of x, in the range 0 to π.

Atan returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

Sin, *cos* and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to **ERANGE.**

Tan returns **HUGE** for an argument which is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets *errno* to **ERANGE.**

Arguments of magnitude greater than 1.0 cause *asin* and *acos* to return 0 and to set *errno* to **EDOM.** In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );
```

DESCRIPTION

tsearch, tfind, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Tsearch is used to build and access the tree. **Key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **Rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

Tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. **Root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration

data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second, or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <stdio.h>

struct node {  /*pointers to these are stored in the tree*/
        char *string;
        int length;
};
char string_space[10000];          /*space to store strings*/
struct node nodes[500];            /*nodes to store*/
struct node *root = NULL;     /*this points to the
root*/

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0; node_compare( );

        while (gets(strptr) != NULL && i++ < 500) {
                /*set node*/
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /*put node into the tree*/
                (void) tsearch((char *)nodeptr, (char **)
                &root,
                        node_compare);
                /*adjust pointers so we don't overwrite
                tree*/
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk((char *)root, print_node);
}
/*
        This routine compares two nodes, based on an
```

```
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
        return strcmp(((struct node *)node1)->string,
        ((struct node *) node2)->string;
}
/*
        This routine prints out a node, the
        first time twalk encounters it.
*/
void
print_node(node, order, level)
char **node;
VISIT order;
int level;
{
        if (order == preorder  order == leaf) {
                (void)printf("string = %20s, length =
                %dxn",
                        (*((struct                node
                        **)node))->string,
                        (*((struct                node
                        **)node))->length);
        }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tfind* and *tdelete* if **rootp** is NULL on entry. If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The **root** argument to *twalk* is one level of indirection **less** than the **rootp** arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *Tsearch* uses preorder, postorder, and endorder to respectively refer to visiting a node before any of its children, after its left child and before its right, and after both children. The alternate nomenclature uses preorder, inorder, and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ungetc – push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc* call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

In order that *ungetc* perform correctly, a read statement must have been performed prior to the call of the *ungetc* function. *Ungetc* returns **EOF** if it can't insert the character. In the case that *stream* is *stdin*, *ungetc* will allow exactly one character to be pushed back onto the buffer without a previous read statement.

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

Vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates the use of *vfprintf* to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
/*  error should be called like
/*  error(function_name, format, arg1, arg2...);  */
/*  VARARGS  */
void
error(va_alist)
/*  Note that the function_name and format arguments cannot be
/*  separately declared because of the definition of varargs.  */
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /*  print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /*  print out remainder of message  */
        (void)vfprintf(stderr, fmt, args);
        va_end(args);
        (void)abort( );
```

VPRINTF (3S)

```
}
```

SEE ALSO

printf(3S), varargs(5).

NAME

wind – creates and places a window

SYNOPSIS

```
#include <wind.h>

int
wind(type, height, width, flags, pfont)
int type, height, width;
short flags;
char *pfont[];
```

DESCRIPTION

Wind creates a window that is of *height* by *width* characters and loads the window with the fonts listed in *pfont*. Unlike *wcreate*, *wind* does not need specific coordinates to create a window but creates one of three types of windows: W_POPUP (inside the existing window), W_SON (overlapping the existing window), or W_NEW (a new window that tries not to overlap the existing window). The three types of windows are described in detail below.

W_POPUP makes the new window live "inside" the window *wncur*. Inside is defined as completely within and centered. Overflow goes down and to the right if possible.

W_SON makes the new window on the lower right corner if possible. The overlap is determined based on the size of the window *wncur*.

W_NEW creates a window in a new part of the display, avoiding existing windows. This is slower and should only be used where necessary.

EXAMPLES

```
menu_t *m;
int height, width, windop, wn;
        .
        .
        .
    height = 5;
    width = 10;
    if ( m -> m_flags & M_WINSON )
        windop = W_SON;
    else if ( m -> m_flags & M_WINNEW )
        windop = W_NEW;
    else
        windop = W_POPUP;
    wn = wind (windop, height, width, M_BORDFLAGS, 0 );
    m -> m_win = wn;
        .
        .
```

SEE ALSO

form(3T), menu(3T), tam(3T), window(7).

DIAGNOSTICS

If *wind* returns a positive number, the number is the window

number. A negative number indicates an error, as defined in **wind.h**.

BUGS

Pfont is currently ignored.

NAME

wrastop – pixel raster operations for bitmap displays

SYNOPSIS

```
#include <sys/window.h>
int wrastop(w, srcbase, srcwidth, dstbase, dstwidth,
            srcx, srcy, dstx, dsty, width, height,
            srcop, dstop, pattern)
int w;
unsigned short *srcbase, *dstbase, *pattern;
unsigned short srcwidth, dstwidth;
unsigned short srcx, srcy, dstx, dsty;
char srcop, dstop;
```

DESCRIPTION

The *wrastop* routine provides user programs with direct access to a window's pixel data. This "raster operation" is controlled by the arguments which include both source and destination operators:

```
/* rastop source operators */
#define SRCSRC      0   /* source */
#define SRCPAT      1   /* pattern */
#define SRCAND      2   /* source and pattern */
#define SRCOR       3   /* source or pattern */
#define SRCXOR      4   /* source xor pattern */

/* rastop destination operators */
#define DSTSRC      0   /* srcop(src) */
#define DSTAND      1   /* srcop(src) and dst */
#define DSTOR       2   /* srcop(src) or dst */
#define DSTXOR      3   /* srcop(src) xor dst */
#define DSTCAM      4   /* not(srcop) and dst */
```

W is the window identifier for the window to be accessed (see *tam*(3T) for more information on window identifiers). The *srcbase* and *dstbase* arguments determine the memory addresses of the source and destination planes. *Srcbase* and *dstbase* may point to the address of the first short of an arbitrarily-sized array of shorts. Each row of pixels consists of *srcwidth* (or *dstwidth*) number of bytes from this array. Thus, the first pixel row exists from *srcbase* to ((char *)*srcbase*) + *srcwidth*. Within each short, the least significant bit is the left-most when displayed on the screen.

Alternatively, *srcbase* and/or *dstbase* may contain 0, in which case the source or destination is assumed to be the window specified by the first arg to the call. The caller need not supply any value for the *srcwidth* if *srcbase* is 0, nor *dstwidth* if *dstbase* is zero. It is therefore possible to perform raster operations from user space to user space, user space to screen, screen to user space, or screen to screen.

The *srcx*, *srcy*, *dstx*, and *dsty* parameters contain pixel addresses within the specified pixel plane. 0,0 is always the upper-left-hand corner of the display. Note that raster operations are completely

aware of the problems associated with overlapping rectangles: the memory operations will be done front to back or back to front as necessary.

The *width* and *height* parameters give the rectangle's width and height in pixels.

The *srcop* (source operation) and *dstop* (destination operation) fields together determine the algorithm which will be applied to the two rectangles. The basic behavior of *rastop* conforms to the following vector description:

dst = dstop(srcop(src,pattern))

where *srcop* and *dstop* are vector functions. There are five source operations. SRCSRC is the identity function whose value is the unmodified source rectangle itself. SRCPAT's value is that of the "pattern" (see below) and bears no relationship to the source. SRCOR is the inclusive OR of the source and the pattern; SRCAND, the AND; SRCXOR, the exclusive OR.

DSTSRC is the identity function, returning the result of the source operation unchanged. DSTAND is the AND of the destination with the result of the source, DSTOR is the inclusive OR, and DSRXOR the exclusive OR. DSTCAM AND's the one's-complement of the source operation into the destination. DSTCAM is the inverse of DSTOR: where DSTOR would turn on pixels, DSTCAM will turn them off.

The *pattern* field is required for SRCPAT, SRCAND, SRCOR, and SRCXOR operations only. It points to an array of 16 X 16 pixels arranged as 16 consecutive shorts. As with source and destination rectangles, the LSB of the first short in the vector corresponds to the upper-left-hand pixel of the pattern. *Patterns* are automatically aligned with the destination.

In addition to the *wrastop* function, there are four pre-defined *patterns*: **patblack** (all zeros), **patwhite** (all ones), **patgray** (halftone), and **patltgray** (light gray). To reference these *patterns*, the calling program should define these *patterns* as external unsigned short arrays (unsigned short patblack[ ]).

If the *pattern* field is set to 0, the operation will take place as if **patblack** was specified.

Note that *wrastop* always refreshes the specified window before executing to force any character operations to occur in correct time order.

FILES

/usr/include/sys/window.h

SEE ALSO

tam(3T), window(7).

DIAGNOSTICS

Wrastop returns 0 on success, -1 on failure with *errno* set to the error number. Any attempt to issue a *wrastop* call on a non-bitmap display will result in a return of -1 with *errno* left to its previous value.

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

Files on the UNIX PC cannot be larger than 1 megabyte in size.

References of the type *name*(1M) refer to entries found in Section 1M of the *UNIX PC UNIX System User's Manual*.

NAME

a.out – common assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

The file name **a.out** is the output file from the assembler *as*(1) and the link editor *ld*(1). Both programs will make *a.out* executable if there were no errors in assembling or linking and no unresolved external references.

A common object file consists of a file header, a UNIX system header, a table of section headers, relocation information, (optional) line numbers, a symbol table, and a string table. The order is given below.

File header.
UNIX header.
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Section 1 line numbers.
...
Section n line numbers.
Symbol table.
String table.

The last three parts (line numbers, symbol table and string table) may be missing if the program was linked with the **–s** option of *ld*(1) or if the symbol table and relocation bits were removed by *strip*(1). Also note that the relocation information will be absent if there were no unresolved external references after linking. The string table exists only if the symbol table contains symbols with names longer than eight characters.

The sizes of each section (contained in the header, discussed below) are in bytes and are even.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. The text segment begins at location 0x0000 in the core image. The header is never loaded except for magic 0413 files created with the **–F** option of *ld*(1). If the magic number (the first field in the operating system header) is 407 (octal), it indicates that the text segment is not to be write-protected or shared, so the data segment will be contiguous with the text segment. If the magic number is 410 (octal), the

data segment and the text segment are not writable by the program; if other processes are executing the same **a.out** file, they will share a single text segment.

Magic number 413 (octal) is the same as 410 (octal), except that 413 (octal) permits demand paging. Both the **-z** and **-F** options of the loader *ld*(1) create **a.out** files with magic numbers 0413. If the **-z** option is used, both the text and data sections of the file are on 1024-byte boundaries. If the **-F** option is used, the text and data sections of the file are contiguous. Loading a single 4096-byte page into memory requires 4 transfers of 1024 bytes each for **-z**, and typically one transfer of 4096 bytes for **-F**. Thus **a.out** files created with **-F** can load faster and require less disk space.

The stack begins at the end of memory and grows towards lower addresses. The stack is automatically extended as required. The data segment is extended only as requested by the *brk*(2) system call.

The value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text involves a reference to an undefined external symbol, the storage class of the symbol-table entry for that word will be marked as an "external symbol," and the section number will be set to 0. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

File Header

The format of the **filehdr** header is:

```
struct filehdr
{
        unsigned short  f_magic;   /* magic number */
        unsigned short  f_nscns;   /* number of sections */
        long            f_timdat;  /* time and date stamp */
        long            f_symptr;  /* file ptr to symtab */
        long            f_nsyms;   /* # symtab entries */
        unsigned short  f_opthdr;  /* sizeof(opt hdr) */
        unsigned short  f_flags;   /* flags */
};
```

UNIX System Header

The format of the UNIX system header is:

```
typedef struct aouthdr
{
        short   magic;      /* magic number */
        short   vstamp;     /* version stamp */
        long    tsize;      /* text size in bytes, padded */
        long    dsize;      /* initialized data (.data) */
        long    bsize;      /* uninitialized data (.bss) */
        long    entry;      /* entry point */
        long    text_start; /* base of text used for this file */
        long    data_start; /* base of data used for this file */
```

```
} AOUTHDR;
```

Section Header

The format of the **section** header is:

```
struct scnhdr
{
        char            s_name[SYMNMLEN];/* section name */
        long            s_paddr;    /* physical address */
        long            s_vaddr;    /* virtual address */
        long            s_size;     /* section size */
        long            s_scnptr;   /* file ptr to raw data */
        long            s_relptr;   /* file ptr to relocation */
        long            s_lnnoptr;  /* file ptr to line numbers */
        unsigned short  s_nreloc;   /* # reloc entries */
        unsigned short  s_nlnno;    /* # line number entries */
        long            s_flags;    /* flags */
};
```

Relocation

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format:

```
struct reloc
{
        long        r_vaddr;    /* (virtual) address of reference */
        long        r_symndx;   /* index into symbol table */
        short       r_type;     /* relocation type */
};
```

The start of the relocation information is *s_relptr* from the Section Header. If there is no relocation information, *s_relptr* is 0.

Symbol Table

The format of the **symbol table** header is:

```
#define  SYMNMLEN  8
#define  FILNMLEN  14
#define  SYMESZ    18          /* the size of a SYMENT */

struct syment
{
    union                       /* get a symbol name */
    {
        char        n_name[SYMNMLEN];/* name of symbol */
        struct
        {
            long    _n_zeroes;  /* ==0L if in string table */
            long    _n_offset;  /* location in string table */
        }_n_n;
        char        *_n_nptr[2]; /* allows overlaying */
    }_n;
    unsigned long   n_value;    /* value of symbol */
    short           n_scnum;    /* section number */
    unsigned short  n_type;     /* type and derived type */
    char            n_sclass;   /* storage class */
```

```
    char            n_numaux;   /* number of aux entries */
};

#define     n_name      _n._n_name
#define     n_zeroes    _n._n_n._n_zeroes
#define     n_offset    _n._n_n._n_offset
#define     n_nptr      _n._n_nptr[1]
```

Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows:

```
union auxent {
    struct {
        long x_tagndx;
        union {
            struct {
                unsigned short  x_lnno;
                unsigned short  x_size;
            } x_lnsz;
            long x_fsize;
        } x_misc;
        union {
            struct {
                long x_lnnoptr;
                long x_endndx;
            } x_fcn;
            struct {
                unsigned short  x_dimen[DIMNUM];
            } x_ary;
        } x_fcnary;
        unsigned short x_tvndx;
    } x_sym;

    struct {
        char x_fname[FILNMLEN];
    } x_file;

    struct {
        long        x_scnlen;
        unsigned short  x_nreloc;
        unsigned short  x_nlinno;
    } x_scn;

    struct {
        long        x_tvfill;
        unsigned short  x_tvlen;
        unsigned short  x_tvran[2];
    } x_tv;
};
```

Indexes of symbol table entries begin at *zero*. The start of the symbol table is *f_symptr* (from the file header) bytes from the beginning of the file. If the symbol table is stripped, *f_symptr* is 0. The string table (if one exists) begins at *f_symptr* + (*f_nsyms*

SYSEMZ) bytes from the beginning of the file.

SEE ALSO

as(1), cc(1), ld(1), brk(2), filehdr(4), ldfcn(4), linenum(4), reloc(4), scnhdr(4), syms(4).

NAME

adf – application data format

DESCRIPTION

The Application Data Format (ADF) is a file format designed to support data interchange between different applications. ADF is an ASCII format with predefined *keywords* and *tokens*.

ADF consists of five basic data types: **file operations**, **text**, **numeric**, **graphic**, and **spreadsheet**, along with their subtypes. In addition to the basic data types, there are higher order data types, such as **tuples**, **tables**, and **groups**, which are collections of basic data types.

Each data type starts with a *keyword*, which is followed by a number of arguments depending on the keyword. Each keyword always appears at the beginning of a line, and is always terminated with white space (ASCII tab or space) or a new line character. All keywords accept either type of terminator. White space at the beginning of the line is ignored.

In some cases, the arguments of keywords are also keyed, and in this case the reserved words are used to introduce argument values which are referred to as *tokens*. Tokens always appear at the beginning of the line, and are terminated with white space or new line characters.

Keywords that are followed by arguments of indefinite length or number use braces to delimit the arguments that belong to the keyword. These braces may be nested, and the matching end brace terminates the keyword definition.

Any line beginning with the pound sign (#) character is ignored. In this way comments may be embedded in an ADF file. An exception to this rule is in the **text** data type, where a pound sign may end up at the beginning of a line.

All lines are required to be less than 80 characters in length.

The ADF file begins with the ADF keyword, which identifies it as an ADF file, followed by the VERSION keyword and the APPLICATION keyword. The initial version of ADF is VERSION 1.0. The APPLICATION keyword identifies the source of the ADF file.

Following this header are an arbitrary number of data items, and the file ends with the EOF keyword.

File Operations

ADF supports the moving and copying of entire files and file folders.

The FILOP data type has the following format:

```
FILOP <file operation> {
        SOURCE      <first source file>
        NAME        <first destination name>
        . . . .
        . . . .
        SOURCE      <last source file>
```

```
        NAME        <last destination name>
        }
```

The file operations which are currently supported are COPY and MOVE.

There is one SOURCE keyword for each file which is to be copied or moved. The argument to the SOURCE keyword is the full path name of the file or folder to be copied or moved.

The NAME keyword is optional and is necessary if the name of the destination file is different than the name of the source file. The NAME keyword is especially helpful for applications which copy files into a temporary directory before they are finally copied or moved to their ultimate destination.

Here is an example of an ADF file which is used to copy attachments for Electronic Mail:

```
VERSION 1.0
APPLICATION Electronic Mail
FILOP MOVE {
        SOURCE /tmp/EMAAAa00273
        NAME homedir1
        SOURCE /tmp/EMBAAa00273
        NAME insfmla:S
        SOURCE /tmp/EMCAAa00273
        NAME movepasswd
        SOURCE /tmp/EMDAAa00273
        NAME relativeatt
        SOURCE /tmp/EMEAAa00273
        NAME addr.c
        }
EOF
```

Text Data Types

The TEXT keyword takes a single argument, an ASCII coded text string. White space following the TEXT keyword on the same line is ignored. The **text** data type is a stream of bytes, together with associated attribute and font information. Non-ASCII characters and attribute information are embedded in the text string via the backslash character (\). The text string is terminated with the \EOT\ code.

All ASCII characters except backslash (\) require no interpretation. The backslash character introduces one of the following codes:

Code	Meaning
\\	Converts to a single backslash
\<decimal #>\	Inserts a byte of the specified value. The legal range is 0 - 239.

\<new line>\	Nothing. Used to embed new lines for formatting purposes. Un-escaped new lines are only used for paragraph ends.
\IND\	Indent character
\CEN\	Center character
\RB\	Required backspace
\HS\	Hard space
\OH\	Optional hyphen
\HH\	Hard hyphen
\HI\	Hanging indent
\BB\	Begin block (keep block)
\EB\	End block
\PN\	Page number
\EOP\	End of page
\HP\	Hard page end
\BF\	Begin field name
\EF\	End field name
\RS\	Record separator
\EOT\	End of text string
\UL\	Underline start
\WU\	Word underline start
\DU\	Double underline start
\US\	Underline stop
\BL\	Bold start
\BS\	Bold stop
\MI\	Mark insert
\MIS\	Mark insert stop
\MD\	Mark delete (strike-thru)
\MDS\	Mark delete stop
\SUP\	Superscript
\SUS\	Superscript stop
\SUB\	Subscript
\SBS\	Subscript stop
\FONT:<NAME>\	Select font <NAME> (predefined)
\CSIZE:<n>\	Select character size <n> (point size)
\COLOR:<n>\	Select color <n> (1-8)

A font is a mapping of the text stream byte values onto displayable characters. The form that this mapping takes depends on the output devices supported by the particular system. The font also contains pitch information and the widths of the individual characters, in the case of a proportionally spaced font.

For transfers between applications running on different machines, the font information for all of the referenced fonts must be included in the ADF file.

Here are some sample text data items:

TEXT Short string\EOT\

```
TEXT
This is a sample paragraph. Notice that \UL\all\US\ \
of the new lines are escaped except for the one at the \
end of the paragraph, and \BL\all\BS\ of the spaces \
between words are present.
\EOT\
```

The LABEL data type is a subtype of the **text** data type. It has the following format:

```
LABEL <x coordinate> <y coordinate> {
        LABELORIGIN   <label origin number>
           .          <optional keywords>
           .
           .
        TEXT          <text>\EOT\
        }
```

Coordinates are floating point numbers in the range 0-100. The *label origin number* is an integer between 0 and 8 (inclusive), and refers to the justification of the label.

The *optional keywords* are used to define attributes of particular types of LABELs, such as graphic LABELs or spreadsheet LABELs, and are described under the appropriate data types.

The TEXT data is a watered down version of the **text** data type previously described. The special codes defined for the **text** data type are all legal, but are mostly ignored.

Numeric Data Types

The following numeric data types are defined:

Keyword	Meaning
INT	Integer
FLOAT	Floating point number
TIME	Year/Month/Day or Hour/Minute/Second
DURATION	Hours:Minutes

Optionally following the keyword are the following tokens:

Token	Meaning
JU <x>	Justification mode. <x> = L, R, C, or D for left, right, center, or decimal justify.

FORMAT <y> Display format, where <y> = I, F, E, M, D, or T for integer, floating, exponential (scientific notation), money, date, or time. F is followed by the number of digits to the left and right of the decimal point, e.g. F3.2.

Finally there is the number, coded as an ASCII decimal string, optionally using an exponential notation.

A series of numbers, separated by white space and terminated by an unescaped new line character, may also follow at this point.

Here are some sample numeric data items:

```
INT     25

INT     -37 43 376 8892\
        96 248 -6230 7185

FLOAT
        JU D
        FORMAT F7.3
                29.451 0.675E5 4

DURATION    1:07
```

Graphic Data Types

The basic graphic data types are OBJECTs and LABELs. The LABEL data type is a graphical incarnation of the LABEL data type previously described, while the OBJECT data type is a collection of lines, rectangles, and text that makes up a graphics object (a bar chart, for example).

The graphic LABEL data type has the following format:

```
LABEL <x coordinate> <y coordinate> {
        LABELORIGIN   <label origin number>
        CSIZE         <point size>
        FONT          <font name>
        COLOR         <color number>
        STYLE         <style name>
        TEXT          <text>\EOT\
        }
```

The tokens following the LABEL keyword must all be specified exactly once for each LABEL. The *label origin number* is an integer between 0 and 8 (inclusive), and refers to the justification of the label within its box. The *font name* must be the name of a defined system font. The *color number* is an integer between 1 and 8 (inclusive).

The OBJECT data type has the following format:

OBJECT <x scale> <y scale> <x translation> <y translation> {

DRAWINGMODE	<vector drawing mode>
LINETYPE	<line drawing pattern>
MOVE	<x coordinate> <y coordinate>
DRAW	<x coordinate> <y coordinate>
RECTANGLE	<xlow> <ylow> <xhigh> <yhigh> <pattern>
POLYGON	<n><x[0]><y[0]> ... <x[n]><y[n]><pattern>
LABELORIGIN	<label origin number>
CSIZE	<point size>
FONT	<font name>
COLOR	<color number>
STYLE	<style name>
TEXT	<text>\EOT\

}

The OBJECT data type can (and typically does) contain multiple instances of the above tokens. A single OBJECT, like a bar chart or a pie chart, will contain lots of lines, rectangles, polygons, and text fragments. The matching end brace terminates the object definition.

The tokens that set attributes (e.g. DRAWINGMODE or FONT) affect subsequently defined tokens until overridden by another instance of the attribute setting token. If all of the text in a given object is of the same COLOR, for example, the color only needs to be specified once.

Coordinates are again floating point numbers in the range 0-100. *X* scale and *y* scale are the scaling factors in the x and y dimensions for the object (floating point numbers in the range 0-1). *X translation* and *y translation* are offsets from (0,0) for the object, also in the range 0-100.

Vector drawing mode and *line drawing pattern* are small positive integers. *Pattern* (in RECTANGLE and POLYGON) is an integer for the fill pattern, and *n* (in POLYGON) is the number of vertices of the polygon.

Spreadsheet Data Types

The basic spreadsheet data types are TEXTs, LABELs, VALUEs, and FORMULAs. The TEXT data type is a watered down version of the **text** data type previously described. The LABEL data type is a spreadsheet incarnation of the LABEL data type previously described. The VALUE data type is a spreadsheet version of the FLOAT data type. The FORMULA data type has all of the attributes of the FLOAT data type, but in addition it has the text of a formula as one of its attributes.

The TEXT data type has the following format:

TEXT <text>\EOT\

The special codes defined for the **text** data type are all legal, but are mostly ignored.

The LABEL data type has the following format:

```
LABEL <x coordinate> <y coordinate> {
        LABELORIGIN     <label origin number>
        REPEATING
        NAME            <name>\EOT\
        LOCK
        INVISIBLE
        TEXT            <text>\EOT\
        }
```

The tokens following the LABEL keyword must all be specified exactly once for each LABEL. The only required token is the TEXT token. The *label origin number* is an integer equal to 1, 4, or 7, and describes the justification of the label. Left justified labels use 1; center justified labels use 4; and right justified labels use 7 as their *label origin number*. If no LABELORIGIN keyword is present, the label is assumed to be left justified. A repeating label is identified by including the REPEATING keyword. The NAME token is used to specify a name which may contain up to 15 characters. The LOCK keyword is used to specify the local lock status of the label. If no LOCK keyword is present, the lock status is determined by the global lock status. The INVISIBLE keyword is used to indicate a label which is invisible (non-displaying). The TEXT token identifies a watered down version of the **text** data type previously described. The special codes defined for the **text** data type are all legal, but are mostly ignored.

The VALUE data type has the following format:

```
VALUE <x coordinate> <y coordinate> {
        NAME            <name>\EOT\
        LOCK
        INVISIBLE
        FORMAT          <format type>
        DECIMALS        <number of decimal places>
        <integer or floating point number>
        }
```

The tokens following the VALUE keyword must all be specified exactly once for each VALUE. The only required token is the *integer or floating point number* token. The NAME token is used to specify a name which may contain up to 15 characters. The LOCK keyword is used to specify the local lock status of the value. If no LOCK keyword is present, the lock status is determined by the global lock status. The INVISIBLE keyword is used to indicate a value which is invisible (non-displaying). The FORMAT token identifies one of the following formats:

Format	Meaning
Comma	Fixed format with commas.
Dollars and Cents	Positive numbers are preceded with dollar sign. Negative numbers are enclosed in parentheses. Uses commas.
Financial	Negative numbers are in parentheses.
Fixed	Fixed format without commas.
Percent	Per cent.
Scientific	Scientific notation.

If no FORMAT token is present, the format is determined by the global format. The DECIMALS token specifies the number of characters to be displayed following the decimal point. If no DECIMALS token is present, the number of decimal places is determined by the global decimal places.

The FORMULA data type has the following format:

```
FORMULA <x coordinate> <y coordinate> {
        NAME                <name>\EOT\
        LOCK
        INVISIBLE
        FORMAT              <format type>
        DECIMALS            <number of decimal places>
        TEXT                <text>\EOT\
        <integer or floating point number>
        }
```

The tokens following the FORMULA keyword must all be specified exactly once for each FORMULA. The only required tokens are *integer or floating point number* and TEXT. With the exception of the TEXT token, a FORMULA has all of the same tokens that a VALUE has.

The TEXT token is used to specify the text of the formula. The exact syntax of formula text is determined by the FORMULATYPE keyword in the TABLEDEF data type. Applications which are able to understand the text of a formula should parse the formula, and then store the formula in their own internal format. The value of the formula should be determined by having the application evaluate the formula, not by the number in *integer or floating point number*. Applications which do not understand the text of a formula should ignore the formula text and treat this data type like a VALUE data type.

Compound Data Types

The data types in this section are composed of lists of the already defined basic data types. The following compound data types are defined:

Keyword	Meaning
SCHEMA	Define fields for the TUPLE data type
TUPLE	A collection of data items in the same form as a preceding SCHEMA
TABLEDEF	Table layout definition
TABLE	A rectangular array of data items
GROUP	A collection of graphics objects

The SCHEMA keyword is followed by a series of FIELD*n* tokens, one for each field being defined. Following each FIELD*n* token are NAME, TYPE, and SIZE tokens. These tokens are optional except for the TYPE token, which is required. For example:

Keyword/Token	Meaning
SCHEMA {	
FIELD1	Start definition for field one
NAME <a>	<a> is an ASCII text string which is the field name
TYPE <t>	<t> is the data type of the field. Must be one of the previously defined basic data types.
SIZE <n>	<n> is the field length in bytes
FIELD2	Start definition for field two
.	
.	
.	
}	

The TUPLE keyword is followed by a series of arguments, which are data items. The number of data items following TUPLE is the same as the number of fields defined in the preceding SCHEMA. Numeric data items can be typed in without their keyword, but other data items (text in particular) must include their keyword. The matching end brace terminates the TUPLE. The following is a TUPLE with one text field and two numeric fields:

```
TUPLE {
        TEXT Sample string \EOT\
        25 43.671
        }
```

The TABLEDEF keyword takes two arguments, the width and height of the table. It is optionally followed by a series of tokens which provide default information for the rows and columns of the table. It also contains information necessary to parse the formulas of the table.

The TABLEDEF data type has the following format:

```
TABLEDEF <width> <height> {
   COORDINATES    <absolute column> <absolute row>
   GLOBALWIDTH    <width>
   GLOBALFORMAT   <format type>
   GLOBALDECMLS   <number of decimal places>
   GLOBALLOCK
   FORMULATYPE    <formula type>
   WIDTH          <relative column> <width>
          .
          .
          .
   WIDTH          <relative column> <width>
   }
```

All tokens following the TABLEDEF keyword, except the WIDTH token, must be specified exactly once. There are no required tokens. The COORDINATES token specifies the location of the upper left corner of the table being moved or copied relative to the (perhaps larger) table from which it was moved or copied. The GLOBALWIDTH token specifies the default width of each cell of the table. The available formats are the same as those discussed in the section on the VALUE data type. The GLOBALDECMLS token specifies the default number of decimal places. The GLOBALLOCK token is used to specify the default lock status of each cell of the table. If this token is not present, it is assumed that cells are not locked. The FORMULATYPE token specifies the syntax of the formula text. Possible values for this token are **Lotus**, **Multiplan**, and **Supercomp20**. The WIDTH token is used to make a column have a different width than the default width. The column number is a relative column number and is 0 based so that **WIDTH 0 7** means that the first column of the table has a width of 7.

The following is an example of a TABLEDEF data type:

```
TABLEDEF 7 7 {
        COORDINATES 0 0
        GLOBALWIDTH 8
        GLOBALDECMLS 0
        FORMULASTYLE Lotus
        WIDTH 1 11
        WIDTH 5 9
        }
```

The TABLE keyword is used to introduce a rectangular array of data items in the previously defined table format. The number of data items is the same as the number of columns times the number or rows in the table. Numeric data items can be entered without their keywords, but other data items (text in particular)

must include their keywords. Coordinates can be included with both numeric and text data items to output a sparce matrix. The matching end brace terminates the table.

The following is a sample table:

```
TABLE {
        TEXT \EOT\
        LABEL 1 1 {
                LABELORIGIN 1
                TEXT
Year\EOT\
        }
        25 32 17 74
        FLOAT 6 1 {
                NAME pi\EOT\
                LOCK
                INVISIBLE
                FORMAT Scientific
                DECIMALS 4
3.1415975
        }
        FORMULA 7 1 {
                TEXT SUM(E4 + D5)\EOT\
                FORMAT Financial
                DECIMALS 0
1266.666654
        }
```

The GROUP keyword introduces a collection of graphic entities (OBJECTs and LABELs). It is used as follows:

```
GROUP {
        OBJECT ... {
                .
                .
                .
                }
        LABEL ... {
                .
                .
                .
                }
        }
```

The matching end brace terminates the group.

Formatting Data Types

The data types in this section are used to specify general formatting characteristics of text. The following formatting data types are defined:

Keyword	Meaning
PAGE	Define page layout
PARAGRAPH	Define paragraph formatting

The PAGE keyword is used to define a page layout. The attributes of the page that can be set using this keyword are the page size, margins, and header and footer text. These characteristics can be associated with all pages, this page only, even numbered pages, or odd numbered pages.

The following tokens can follow the PAGE keyword, up to the matching end brace:

Token	Meaning
EVEN	The following tokens apply to even numbered pages
ODD	The following tokens apply to odd numbered pages
FIRST	The following tokens apply to the current page only
HEIGHT <n>	<n> is the paper height (in 240ths)
WIDTH <n>	<n> is the paper width (in 240ths)
TM <n>	<n> is the top margin (in 240ths)
BM <n>	<n> is the bottom margin (in 240ths)
OFFSET <n>	<n> is the offset to the left margin (in 240ths)
PITCH <n>	<n> is the horizontal spacing (in 240ths-0 is proportional)
PN <n>	<n> is the initial page number
HEADER	This token is followed by data items, up to a matching end brace, which constitute the header
FOOTER	This token is followed by data items, up to a matching end brace, which constitute the header

The PARAGRAPH keyword is used to set paragraph formatting characteristics. The attributes of the paragraph that can be set using this keyword are the left and right text margins, the line spacing, paragraph justification mode, and tab stops.

The following tokens can follow the PARAGRAPH keyword:

Token	Meaning
MARGIN <l> <r>	<l> and <r> are the left and right margins (in columns).
JU <x>	Justification mode. <x> = L, R, C, or J for left, right, center, or full (both left and right) justify.
LS <n>	<n> is the interline spacing (in 240ths).
TAB <n> <x>	<n> is the column number of the tab stop, and <x> is the tab type (L, R, C, D, or P for left, right, center, decimal align, or period leader). Multiple tab stops can follow, separated by commas, terminated with an unescaped new line.

SEE ALSO

paste(3T).

NAME

ar – common archive file format

DESCRIPTION

The archive command *ar*(1) is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define  ARMAG      "!<arch> n"    /* magic string*/
#define  SARMAG 8                  /*length of magic string*/
```

Each archive which contains common object files (see *a.out*(4)) includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define ARFMAG      "` n"    /*header trailer string*/

struct   ar_hdr              /*file member header*/
{
        char    ar_name[16];    /*'/' terminated file member name */
        char    ar_date[12];    /* file member date */
        char    ar_uid[6];      /* file member user identification*/
        char    ar_gid[6];      /* file member group identification*/
        char    ar_mode[8];     /* file member mode (octal)*/
        char    ar_size[10];    /* file member size */
        char    ar_fmag(2);     /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* field is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name[0]** =='/'). The contents of this file are as follows:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: 4 bytes * "the number of symbols."
- The name string table. Length: *ar_size* – (4 bytes * ("the number of symbols" + 1)).

The number of symbols and the array of offsets are managed with *sgetl* and *sputl*. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

ar(1), ld(1), strip(1), sputl(3X), a.out(4).

BUGS

Strip(1) will remove all archive symbol entries from the header. The archive symbol entries must be restored via the **ts** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

NAME

checklist – list of file systems processed by fsck

DESCRIPTION

Checklist resides in directory **/etc** and contains a list of at most 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck*(1M) command.

SEE ALSO

fsck(1M).

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). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The important stuff not detailed therein is the locations of the registers, which are outlined in **/usr/include/sys/reg.h**.

SEE ALSO

sdb(1), setuid(2), signal(2).

NAME

cpio – format of cpio archive

DESCRIPTION

The *header* structure, when the **–c** option of *cpio*(1) is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **–c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(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 *fs*(4)). The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ 14
#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; there is no parent, so **..** has the same meaning as **.**.

SEE ALSO

fs(4).

NAME

filehdr – file header for common object files

SYNOPSIS

#include <filehdr.h>

DESCRIPTION

Every common object file begins with a 20-byte header. The following C **struct** declaration is used:

```
struct filehdr
{
        unsigned short  f_magic ;   /* magic number */
        unsigned short  f_nscns ;   /* number of sections */
        long            f_timdat ;  /* time & date stamp */
        long            f_symptr ;  /* file ptr to symtab */
        long            f_nsyms ;   /* # symtab entries */
        unsigned short  f_opthdr ;  /* sizeof(opt hdr) */
        unsigned short  f_flags ;   /* flags */
} ;
```

F_symptr is the byte offset into the file at which the symbol table can be found. Its value can be used as the offset in *fseek*(3S) to position an I/O stream to the symbol table. The UNIX optional header is always 36 bytes. The valid magic numbers are given below:

```
#define N3BMAGIC      0550    /* 3B20S */
#define NTVMAGIC      0551    /* 3B20S */

#define VAXWRMAGIC    0570    /* VAX writable text segments */
#define VAXROMAGIC    0575    /* VAX readonly sharable text */
                              /* segments */
```

The value in *f_timdat* is obtained from the *time*(2) system call. Flag bits currently defined are:

```
#define F_RELFLG    00001   /* relocation entries stripped */
#define F_EXEC      00002   /* file is executable */
#define F_LNNO      00004   /* line numbers stripped */
#define F_LSYMS     00010   /* local symbols stripped */
#define F_MINMAL    00020   /* minimal object file */
#define F_UPDATE    00040   /* update file, ogen produced */
#define F_SWABD     00100   /* file is "pre-swabbed" */
#define F_AR16WR    00200   /* 16 bit DEC host */
#define F_AR32WR    00400   /* 32 bit DEC host */
#define F_AR32W     01000   /* non-DEC host */
#define F_PATCH     02000   /* "patch" list in opt hdr */
```

SEE ALSO

time(2), fseek(3S), a.out(4).

NAME

font – font file format

DESCRIPTION

A font is a collection of 96 variably-sized graphics. A font exists first on disk as a "font file." Font files are loaded into the kernel via the WIOCLFONT (see *window*(7)) or SYSL_LFONT (see *sys-local*(2)) *ioctl*. Each font file has three sections: first, a header containing information about the font as a whole; second, a 96-entry table describing each of up to 96 characters in the font; and third, a variable number of "miniraster," each an array of 16-bit words containing the pixel definition of each character. The font file format is given below:

```
#define FMAGIC  0616      /* font magic number */
#define FNTSIZE 96        /* size of a font */

struct fntdef
{
        long            ff_magic;       /* magic number */
        unsigned char   ff_flags;       /* flags */
        char            ff_hs;          /* hor spacing */
        char            ff_vs;          /* ver spacing */
        char            ff_baseline;    /* baseline */
        char            ff_dummy[26];   /* padding */
        struct fcdef    ff_fc[FNTSIZE]; /* char defs */
        unsigned short  ff_raster;      /* minirasters */
};

struct fcdef        /* font character definition */
{
        char    fc_hs;  /* horizontal size in bits */
        char    fc_vs;  /* vertical size */
        char    fc_ha;  /* horizontal adjust (signed) */
        char    fc_va;  /* vertical adjust (signed) */
        char    fc_hi;  /* horizontal increment */
        char    fc_vi;  /* vertical increment */
        short   fc_mr;  /* relative mini-raster pointer */
};
```

Each mini-raster is dealt with as 16-bit words; hence it must be word-aligned, and consist of *fc_hs* raster lines each of which contains an integral number of 16-bit data words. The actual position of upper-left corner of miniraster is (*curx* + *fc_ha*, *cury* + *fc_va*). Every word of mini-raster information is stored HIGH byte first, a la mc68000. The low order bit of the first word is the left-most raster point. Bit-0 of the first word thus corresponds to the upper-left corner of the character.

The actual bit pattern of a character is flush left in its mini-raster. The bits to the right of the pattern (i.e. to the right of *fc_hs*) and before the short boundary must be 0. Normally, *fc_va* is negative, thus implying that coordinate (0, 0) is upper left.

Note that this font format is similar to both the MIT "Nu" System font format and the Berkeley "vfont" format.

Note too that fonts which appear in windows must be 12-point.

A variant of a font character (**struct fcdef**) is an icon. The icon structure differs from a font character in that the miniraster information is located immediately after the header information. Icons are used for, among other things, controlling the appearance of the mouse-tracking cursor on the display.

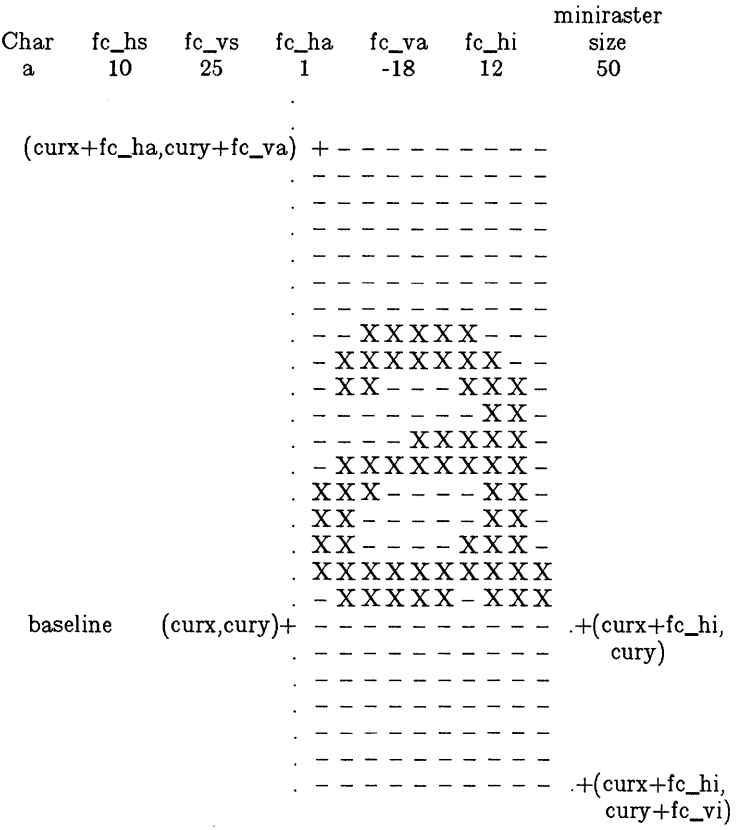
The icon structure is as follows:

```
#define ICONSIZE 64    /* shorts in icon raster area */

struct icon                /* an icon */
{
        char               ic_flags;              /* flags */
        struct fcdef       ic_fc;                 /* font def */
        unsigned short     ic_raster[ICONSIZE];   /* raster data */
};
```

Currently, there are no flags defined for *ic_flags*.

EXAMPLE



FILES

/usr/include/sys/font.h
/usr/lib/wfont/*

BUGS

Older fonts do not have any value specified for the vertical increment (*fc_vi*) as this is a relatively new addition to the font character definition. Instead, these characters are made artificially high to extend over the entire cell height.

SEE ALSO

cfont(1), syslocal(2), window(7), tam(3T).

NAME

file system – format of system volume

SYNOPSIS

#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512 byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *super-block*. The format of a super-block is:

```
/*
 * Structure of the super-block
 */
struct   filsys
{
        ushort    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 */
        short     s_dinfo[4];       /* device information */
        daddr_t   s_tfree;          /* total free blocks */
        ino_t     s_tinode;         /* total free inodes */
        char      s_fname[6];       /* file system name */
        char      s_fpack[6];       /* file system pack name */
        long      s_fill[13];       /* ADJUST to make size */
                                    /* of file system be 512 */
        long      s_magic;          /* magic number to */
                                    /* indicate new file system */
        long      s_type;           /* type of new file system */
};

#define FsMAGIC 0xfd187e20        /* s_magic number */

#define Fs1b    1                 /* 512 byte block */
#define Fs2b    2                 /* 1024 byte block */
```

S_type indicates the file system type. Currently, two types of file systems are supported: the original 512-byte oriented and the new improved 1024-byte oriented. *S_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *FsMAGIC*, the type is assumed to be *Fs1b*, otherwise the *s_type* field is used. In the following description, a block is then determined by the type. For the original 512-byte oriented file system, a block is 512 bytes. For the 1024-byte oriented file system, a block is 1024 bytes or two sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

S_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize*−2 blocks long. *S_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number 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 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; 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 number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

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 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not 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_tinode is the total free inodes available in the file system.

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 read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

S_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long. I-node 1 is reserved for future use. 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. For the format of an inode and its flags, see *inode*(4).

FILES

/usr/include/sys/filsys.h
/usr/include/sys/stat.h

SEE ALSO

fsck(1M), fsdb(1M), mkfs(1M), inode(4).

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on UNIX with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t–8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

* <:t5,10,15 s72:> *

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several UNIX commands correctly interpret the format specification for a file. Among them is *gath*, which may be used to convert files to a standard format acceptable to other UNIX commands.

SEE ALSO

ed(1), newform(1), tabs(1).

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(1M) (see the *UNIX System Administrator's Manual*) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as *\nnn*, where *nnn* is the octal value of the desired character. The various fields are:

label This is the string against which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it needn't be (see below).

initial-flags These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. The flags that *getty* understands are the same as the ones listed in **/usr/include/sys/termio.h** (see *termio*(7) in the *UNIX System Administrator's Manual*). Normally only the speed flag is required in the *initial-flags*. *Getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1M).

final-flags These flags take the same values as the *initial-flags* and are set just prior to *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.

next-label If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty* will search for the entry with *next-label* as its *label* field and set up the terminal for those

settings. Usually, a series of speeds are linked together in this fashion, into a closed set. For instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty* can't find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through *getty* with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

getty(1M), login(1M), termio(7) in the *UNIX System Administrator's Manual.*
ioctl(2).

NAME

group – group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1), passwd(1), crypt(3C), passwd(4).

NAME

inittab – script for the init process

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process */etc/getty* that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

id:rstate:action:process

Each entry is delimited by a newline, however, a backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *gettys* are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one or two characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is to be processed. *Run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field will be processed. When *init* is requested to change *run-levels*, all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20 second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0–6**. If no *run-level* is specified, then the process is assumed to be valid at all *run-levels* **0–6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* (see *init*(1M)) process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that *init* can never enter *run-level* **a**, **b** or **c**. Also, a request for the execution of any of these processes does not change the current *run-level*. Furthermore, a process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in

/etc/inittab is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination and when it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program will not be restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, not wait for its termination, and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, wait for its termination and, when it dies, not restart the process.

powerfail Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR** see *signal*(2)).

powerwait Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault An entry with this *action* is only scanned when *init* is initially invoked. *Init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* will enter *run-level* **6**. Also, the **initdefault** entry cannot specify that *init* start in the *SINGLE USER* state. Additionally, if *init* doesn't find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console. It is expected that this entry will be only used to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh -c 'exec** *command***'**. For this reason, any legal *sh* syntax can appear in the the *process* field. Comments can be inserted with the **:** *#comment* syntax.

FILES

/etc/inittab

SEE ALSO

getty(1M), init(1M) in the *UNIX System Administrator's Manual*. sh(1), who(1), exec(2), open(2), signal(2).

BUGS

On the UNIX PC the **inittab** file is often manipulated by user-level Administration functions performed through the Office. Haphazard modification by the user of **/etc/inittab** can thoroughly confuse this set of functions.

NAME

inode – format of an inode

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>

DESCRIPTION

An i-node for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>**.

```
/* Inode structure as it appears on a disk block. */
struct dinode
{
        ushort  di_mode;    /* mode and type of file */
        short   di_nlink;   /* number of links to file */
        ushort  di_uid;     /* owner's user id */
        ushort  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 */
};
/*
 * the 40 address bytes:
 *     39 used; 13 addresses
 *     of 3 bytes each.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(5).

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(4), types(5).

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

login(1M).

NAME

ldfcn – common object file access routines

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
```

DESCRIPTION

The common object file access routines are a collection of functions for reading an object file that is in VAX or 3B20S (common) object file form. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, defined as **struct ldfile**, declared in the header file **ldfcn.h**. The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function *ldopen*(3X) allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **ldfcn.h** and contain the following information:

LDFILE *ldptr;

TYPE(ldptr) The file magic number, used to distinguish between archive members and simple object files.

IOPTR(ldptr) The file pointer returned by *fopen* and used by the standard input/output functions.

OFFSET(ldptr) The file address of the beginning of the object file; the offset is non-zero if the object file is a member of an archive file.

HEADER(ldptr) The file header structure of the object file.

The object file access functions themselves may be divided into four categories:

(1) functions that open or close an object file

ldopen(3X) and *ldaopen*
open a common object file

ldclose(3X) and *ldaclose*
close a common object file

(2) functions that read header or symbol table information

ldahread(3X)
read the archive header of a member of an archive file

ldfhread(3X)
read the file header of a common object file

ldshread(3X) and *ldnshread*
read a section header of a common object file

ldtbread(3X)
read a symbol table entry of a common object file

(3) functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section.

ldohseek(3X)
seek to the optional file header of a common object file

ldsseek(3X) and *ldnsseek*
seek to a section of a common object file

ldrseek(3X) and *ldnrseek*
seek to the relocation information for a section of a common object file

ldlseek(3X) and *ldnlseek*
seek to the line number information for a section of a common object file

ldtbseek(3X)
seek to the symbol table of a common object file

(4) the function *ldtbindex*(3X) which returns the index of a particular common object file symbol table entry

These functions are described in detail in their respective manual pages.

All the functions except *ldopen*, *ldaopen* and *ldtbindex* return either **SUCCESS** or **FAILURE**, both constants defined in **ldfcn.h**. *Ldopen* and *ldaopen* both return pointers to a **LDFILE** structure.

MACROS

Additional access to an object file is provided through a set of macros defined in **ldfcn.h**. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

```
LDFILE      *ldptr;

GETC(ldptr)
FGETC(ldptr)
GETW(ldptr)
UNGETC(c, ldptr)
FGETS(s, n, ldptr)
FREAD((char *) ptr, sizeof (*ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
```

REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)

See the manual entries for the corresponding standard input/output library functions for details on the use of these macros.

The program must be loaded with the object file access routine library **libld.a**.

CAVEAT

The macro **FSEEK** defined in the header file **ldfcn.h** translates into a call to the standard input/output function *fseek*(3S). **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members!

SEE ALSO

fseek(3S), ldahread(3X), ldclose(3X), ldfhread(3X), ldlread(3X), ldlseek(3X), ldohseek(3X), ldopen(3X), ldrseek(3X), ldlseek(3X), ldshread(3X), ldtbindex(3X), ldtbread(3X), ldtbseek(3X).
Common Object File Format, by I. S. Law.

NAME

linenum – line number entries in a common object file

SYNOPSIS

#include <linenum.h>

DESCRIPTION

Compilers based on *pcc* generate an entry in the object file for each C source line on which a breakpoint is possible (when invoked with the **-g** option; see *cc*(1)). Users can then reference line numbers when using the appropriate software test system (see *sdb*(1)). The structure of these line number entries appears below.

```
struct lineno
{
        union
        {
                long          l_symndx ;
                long          l_paddr ;
        }                     l_addr ;
        unsigned short        l_lnno ;
} ;
```

Numbering starts with one for each function. The initial line number entry for a function has *l_lnno* equal to zero, and the symbol table index of the function's entry is in *l_symndx*. Otherwise, *l_lnno* is non-zero, and *l_paddr* is the physical address of the code for the referenced line. Thus the overall structure is the following:

l_addr	*l_lnno*
function symtab index	0
physical address	line
physical address	line
...	
function symtab index	0
physical address	line
physical address	line
...	

SEE ALSO

cc(1), sdb(1), a.out(4).

NAME

master – master device information table

DESCRIPTION

This file is used by the kernel to obtain device information. The file consists of 3 parts, each separated by a line with a dollar sign ($) in column 1. Part 1 contains device information; part 2 contains names of devices that have aliases; part 3 contains tunable parameter information. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1 contains lines consisting of at least 6 fields, with the fields delimited by tabs and/or blanks:

Field 1: device name (8 characters maximum).

Field 2: device mask (octal)–each "on" bit indicates that the handler exists:
002000 info
001000 release
000400 strategy
000200 print
000100 initialization handler
000040 power-failure handler
000020 open handler
000010 close handler
000004 read handler
000002 write handler
000001 ioctl handler.

Field 3: device line discipline mask (octal)–each "on" bit indicates that the handler exists:
000200 modem interrupt handler
000100 output handler
000040 input handler
000020 ioctl handler
000010 write handler
000004 read handler
000002 close handler
000001 open handler.

Field 4: device type indicator (octal):
000200 allow only one of these devices
000100 suppress count field in the **conf.c** file
000040 suppress interrupt vector
000020 required device
000010 block device
000004 character device
000002 floating vector
000001 fixed vector.

Field 5: handler prefix (4 characters maximum).

Field 6: major device number for block-type device.

Field 7: major device number for character-type device.

Part 2 contains lines with 2 fields each:

Field 1: alias name of device (8 characters maximum).
Field 2: reference name of device (8 characters maximum; specified in part 1).

Part 3 contains lines with 2 or 3 fields each:

Field 1: parameter name (as it appears in description file; 20 characters maximum).
Field 2: parameter name (as it appears in the **conf.c** file; 20 characters maximum).
Field 3: default parameter value (20 characters maximum; parameter specification is required if this field is omitted).

NAME

mnttab – mounted file system table

SYNOPSIS

#include <mnttab.h>

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices, mounted by the *mount*(1M) command, in the following structure as defined by **<mnttab.h>**:

```
struct   mnttab {
         char        mt_dev[10];
         char        mt_filsys[10];
         short       mt_ro_flg;
         time_t      mt_time;
};
```

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the *special file* is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter **NMOUNT** located in **/usr/src/uts/cf/conf.c**, which defines the number of allowable mounted special files.

SEE ALSO

mount(1M), setmnt(1M).

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

The encrypted password consists of 13 characters chosen from a 64 character alphabet (**.**, **/**, **0–9**, **A–Z**, **a–z**), except when the password is null in which case the encrypted password is also null. Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the super-user.)

The first character of the age, *M* say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired will be forced to supply a new one. The next character, *m* say, denotes the minimum period in weeks which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) *M* and *m* have numerical values in the range 0–63 that correspond to the 64 character alphabet shown above (i.e. **/** = 1 week; **z** = 63 weeks). If $m = M = 0$ (derived from the string **.** or **..**) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If $m > M$ (signified, e.g., by the string **./**) only the super-user will be able to change the password.

FILES

/etc/passwd

SEE ALSO

login(1M), passwd(1), a64l(3C), crypt(3C), getpwent(3C), group(4).

BUGS

On the UNIX PC, the **passwd** file is often manipulated by user-level Administration functions performed through the Office.

Haphazard modification by the user of **/etc/passwd** can thoroughly confuse this set of functions.

NAME

phone – phone directory file format

DESCRIPTION

The phone directory (**.phdir**) files created by the Telephone Manager may be used by other programs. These files store telephone directory and user preference information.

Header

The **.phdir** file header occupies the first 256 bytes of the file and is organized in fields as follows. User preference items described below can be set through the Telephone Preferences in the Office.

Field	Description
Magic Number (short)	
Version (short)	
Revision (short)	
No. of Recs. (short)	Number of records contained in the file.
Max. Rec. No. (short)	Largest record number used in the file.
N1 (short)	Pointers to up to 15 records to be included in the Call Screen.
N2 (short)	
N3 (short)	
N4 (short)	
N5 (short)	
N6 (short)	
N7 (short)	
N8 (short)	
N9 (short)	
N10 (short)	
N11 (short)	
N12 (short)	
N13 (short)	
N14 (short)	
N15 (short)	
SLK1 (short)	Pointers to up to 7 records associated with the function keys F1, F3–F8. The Telephone Manager places a call to the associated number when the function key is pressed with Shift. No record is associated with F2 since Shift-F2 displays the Call Screen.
SLK2 (short)	
SLK3 (short)	
SLK4 (short)	
SLK5 (short)	
SLK6 (short)	
SLK7 (short)	
O1 (short)	Pointers to up to 14 records associated with other keyboard keys. Consult the *AT&T UNIX PC Telephone Manager User's Guide* for detailed keyboard information.

O2 (short)
O3 (short)
O4 (short)
O5 (short)
O6 (short)
O7 (short)
O8 (short)
O9 (short)
O10 (short)
O11 (short)
O12 (short)
O13 (short)
O14 (short)

*N1 (long) Character pointers to the field names for the directory entry form.

*N2 (long)
*N3 (long)
*N4 (long)
*N5 (long)
*N6 (long)
*N7 (long)
*N8 (long)
*N9 (long)
*N10 (long)
*N11 (long)

F0 (short) Pointers to the fields that are to be displayed in the directory entry form (user preference item).

F1 (short)
F2 (short)

SEC (short) Number of seconds to delay before logging call (user preference item).

FLAG1 (short) Number of entries allowed in the history list before a warning is issued (user preference item).

FLAG2 (short) Logging of incoming calls enabled or disabled (user preference item).

FLAG3 (short) Logging of outgoing calls enabled or disabled (user preference item).

FLAG4 (short) Invoke Telephone Manager on off-hook enabled or disabled (user preference item).

FLAG5 (short) Display beginning of notes enabled or disabled (user preference item).

FLAG6 (short) Display beginning of history list enabled or disabled (user preference item).

FLAG7 (short) Reserved.

FLAG8 (short) Reserved.

Data

Following the header, bytes 256 to 1023 contain the names assigned to the fields in the directory entry form. Several of these are user preference items.

Bytes 1024 through 1535 contain the master list of the data records. The maximum possible record number is 65536. Bytes 1536 through 15359 contain the index list for the records.

Following the index list are the data records. Each record consists of a string for each field of the directory entry form, delimited by **\n**. When a field is empty, no string appears between the delimiters.

SEE ALSO

phone(7), dial(3C), *AT&T UNIX PC Telephone Manager User's Guide*.

NAME

pnch – file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

NAME

profile – setting up an environment at login time

DESCRIPTION

If your login directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before your session begins; **.profiles** are handy for setting exported environment variables and terminal modes. If the file **/etc/profile** exists, it will be executed for every user before the **.profile**. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
echo "terminal: \c"
read TERM
case $TERM in
    300)            stty cr2 nl0 tabs; tabs;;
    300s)           stty cr2 nl0 tabs; tabs;;
    450)            stty cr2 nl0 tabs; tabs;;
    hp)             stty cr0 nl0 tabs; tabs;;
    745 | 735)      stty cr1 nl1 -tabs; TERM=745;;
    43)             stty cr1 nl0 -tabs;;
    4014 | tek)     stty cr0 nl0 -tabs ff1; TERM=4014; echo "\33;";;
    *)              echo "$TERM unknown";;
esac
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1M), mail(1), sh(1), stty(1), su(1), environ(5), term(5).

BUGS

On the UNIX PC the **profile** file is often manipulated by user-level Administration functions performed through the Office. Haphazard modification by the user of **/etc/profile** can thoroughly confuse this set of functions.

NAME

reloc – relocation information for a common object file

SYNOPSIS

#include <reloc.h>

DESCRIPTION

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format.

```
struct   reloc
{
        long        r_vaddr ;   /* (virtual) address of reference */
        long        r_symndx ;  /* index into symbol table */
        short       r_type ;    /* relocation type */
} ;


/*
 * All generics
 *      reloc. already performed to symbol in the same section
 */
#define  R_ABS                  0

/*
 * 3B generic
 *      24-bit direct reference
 *      24-bit "relative" reference
 *      16-bit optimized "indirect" TV reference
 *      24-bit "indirect" TV reference
 *      32-bit "indirect" TV reference
 */
#define  R_DIR24    04
#define  R_REL24    05
#define  R_OPT16    014
#define  R_IND24    015
#define  R_IND32    016

/*
 * DEC Processors  VAX 11/780 and VAX 11/750
 *
 */
#define R_RELBYTE           017
#define R_RELWORD           020
#define R_RELLONG           021
#define R_PCRBYTE           022
#define R_PCRWORD           023
#define R_PCRLONG           024
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

R_ABS The reference is absolute, and no relocation is necessary. The entry will be ignored.

R_DIR16 A direct, 16-bit reference to a symbol's virtual address.

R_REL16 A "PC-relative," 16-bit reference to a symbol's virtual address. Relative references occur in instructions such as jumps and calls. The actual address used is obtained by adding a constant to the value of the program counter at the time the instruction is executed.

R_IND16 An indirect, 16-bit reference through a transfer vector. The instruction contains the virtual address of the transfer vector, where the actual address of the referenced word is stored.

R_ABS The reference is absolute, and no relocation is necessary. The entry will be ignored.

R_DIR24 A direct, 24-bit reference to a symbol's virtual address.

R_REL24 A "PC-relative," 24-bit reference to a symbol's virtual address. Relative references occur in instructions such as jumps and calls. The actual address used is obtained by adding a constant to the value of the program counter at the time the instruction is executed.

R_OPT16 An optimized, indirect, 16-bit reference through a transfer vector. The instruction contains the offset into the transfer vector table to the transfer vector where the actual address of the referenced word is stored.

R_IND24 An indirect, 24-bit reference through a transfer vector. The instruction contains the virtual address of the transfer vector, where the actual address of the referenced word is stored.

R_IND32 An indirect, 32-bit reference through a transfer vector. The instruction contains the virtual address of the transfer vector, where the actual address of the referenced word is stored.

R_RELBYTE A direct 8 bit reference to a symbol's virtual address.

R_RELWORD A direct 16 bit reference to a symbol's virtual address.

R_RELLONG A direct 32 bit reference to a symbol's virtual address.

R_PCRBYTE A "PC-relative," 8 bit reference to a symbol's virtual address.

R_PCRWORD A "PC-relative," 16 bit reference to a symbol's virtual address.

R_PCRLONG A "PC-relative," 32 bit reference to a symbol's virtual address.

On the VAX processors relocation of a symbol index of -1 indicates that the relative difference between the current segment's start address and the program's load address is added to the relocatable address.

Other relocation types will be defined as they are needed.

Relocation entries are generated automatically by the assembler and automatically utilized by the link editor. A link editor option exists for removing the relocation entries from an object file.

SEE ALSO

ld(1), strip(1), a.out(4), syms(4).

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and

time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one MR number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by newlines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t <type of program>
@f v <program name>
@f i
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default-sid>
@f n
@f j
@f l <lock-releases>
@f q <user defined>
@f z <reserved for use in interfaces>

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **-b** keyletter may be used on the

get command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the %M% identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **-e** keyletter). The **q** flag defines the replacement for the %Q% identification keyword. **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text surrounded by the bracketing lines **@t** and **@T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide in the *UNIX System User's Guide*.

NAME

scnhdr – section header for a common object file

SYNOPSIS

#include <scnhdr.h>

DESCRIPTION

Every common object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The C structure appears below.

```
struct scnhdr
{
        char            s_name[SYMNMLEN]; /* section name */
        long            s_paddr;    /* physical address */
        long            s_vaddr;    /* virtual address */
        long            s_size;     /* section size */
        long            s_scnptr;   /* file ptr to raw data */
        long            s_relptr;   /* file ptr to relocation */
        long            s_lnnoptr;  /* file ptr to line numbers */
        unsigned short  s_nreloc;   /* # reloc entries */
        unsigned short  s_nlnno;    /* # line number entries */
        long            s_flags;    /* flags */
} ;
```

File pointers are byte offsets into the file; they can be used as the offset in a call to *fseek*(3S). If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it. But it can have no relocation entries, line numbers, or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, *s_lnnoptr*, *s_nreloc*, and *s_nlnno* are zero.

SEE ALSO

ld(1), fseek(3S), a.out(4).

NAME

shlib – shared library

DESCRIPTION

UNIX PC UNIX supports a shared library. Its use results in significantly smaller programs, reduced startup times, and better runtime performance. This is accomplished by loading the library with the first program that invokes it. The library is then shared with subsequent programs. Once loaded, the library remains in place until the system is reset.

The shared library contains all the routines traditionally loaded by *-lc*, *-ltam*, and *-ltermlib*.

Use of the shared library requires a change to the makefiles. Source code remains unchanged. The typical makefile link-load line is

```
$(LD) $(LDFLAGS) -o target objects -lc -ltam -ltermlib
```

or

```
$(CC) $(LDFLAGS) -o target objects -ltam -ltermlib
```

should be replaced with

```
$(LD) $(LDFLAGS) $(SHAREDLIB) -o target objects
```

where *target* is the executable and *objects* are the files with the **.o** suffixes. $(SHAREDLIB) is defined in $(MAKEINC)/Makepre.h.

FILES

/lib/shlib
/lib/shlib.ifile
/lib/crt0s.o

SEE ALSO

cc(1), ld(1).

BUGS

Programs that redefine symbols in the shared library cannot use it and must rename the conflicting symbols.
A shared library subroutine cannot contain a breakpoint.

NAME

syms – common object file symbol table format

SYNOPSIS

#include <syms.h>

DESCRIPTION

Common object files contain information to support *symbolic* software testing (see *sdb*(1). Line number entries, *linenum*(4), and extensive symbolic information permit testing at the C *source* level. Every object file's symbol table is organized as shown below.

```
File name 1.
        Function 1.
                Local symbols for function 1.
        Function 2
                Local symbols for function 2.
        ...
        Static externs for file 1.

File name 2.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The C structure is given below.

```
#define SYMNMLEN    8
#define FILNMLEN    14

struct    syment
{
    char              n_name[SYMNMLEN] ;
    long              n_value ;     /* value of symbol */
    short             n_scnum ;     /* section number */
    unsigned short    n_type ;      /* type and derived type */
    char              n_sclass ;    /* storage class */
    char              n_numaux ;    /* number of aux entries */
} ;
```

Meaningful values and explanations for them are given in both **syms.h** and *Common Object File Format*. Anyone who needs to interpret the entries should seek more information in these sources. Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent
{
        struct
        {
                long            x_tagndx;
                union
                {
                        struct
                        {
                                unsigned short  x_lnno;
                                unsigned short  x_size;
                        } x_lnsz;
                        long    x_fsize;
                } x_misc;
                union
                {
                        struct
                        {
                                long    x_lnnoptr;
                                long    x_endndx;
                        }       x_fcn;
                        struct
                        {
                                unsigned short  x_dimen[DIMNUM];
                        }       x_ary;
                }               x_fcnary;
                unsigned short  x_tvndx;
        }       x_sym;
        struct
        {
                char    x_fname[FILNMLEN];
        }       x_file;
        struct
        {
                long    x_scnlen;
                unsigned short  x_nreloc;
                unsigned short  x_nlinno;
        }       x_scn;

        struct
        {
                unsigned short  x_tvlen;
                unsigned short  x_tvran[2];
        }       x_tv;
};
```

Indexes of symbol table entries begin at *zero*.

SEE ALSO

sdb(1), a.out(4), linenum(4).
Common Object File Format by I. S. Law.

NAME

ua – user agent configuration files

DESCRIPTION

The user agent configuration files in the directory **/usr/lib/ua** may be used by other programs. Since these files direct the actions of the user agent, they are protected against inadvertent modification. The files can be changed only by super user editing them in *ed* (or whatever) or by programmatic changes (e.g., an *install* or *remove* script).

The configuration files are:

Suffixes	Defines file types and default actions on files.
Office	Defines initial office objects and their default actions.
Administration	Defines initial administration objects.
Preferences	Defines initial objects for which preferences can be set.
Environment	Defines environment variables.
Comm_pkgs	Provides access to emulator applications through the phone manager.

All the files use a keyword = value syntax. The keyword is always at the beginning of a line, and a newline terminates the value. White space surrounding the keywords is ignored, as are empty lines. Keyword matches are case sensitive, and when an undefined keyword is found, the line is ignored.

To insert a comment, start the line with a pound sign (#).

To define new objects or environment variables, add files of these names in your home directory. User definitions add to the system definitions, and where conflicts occur, the user definitions override the system definitions.

The filenames begin with an upper case letter so that they will appear first in directory listings.

Objects and Actions

Except for Environment, all the files define objects for the user agent and the legal actions (commands) that can be performed on them. The commands for any object must be taken from the set of user agent commands (see below).

Before any objects are defined, the user agent knows about two classes of objects, files and menu objects, and about the commands that operate on them.

The file objects class includes all UNIX files. Initially, there are three objects defined in the files class: directories, executable files, and files.

No menu objects are initially defined. Menu objects are defined in the special files **Office**, **Administration**, and **Preferences**. (The last two files are menu objects in the Office menu.)

The **Suffixes** file defines objects in the file class. These definitions override the default classification of files. That is, if a file is encountered with a defined suffix, it is assumed to be of that type, and whether the file is a directory or is executable is not checked.

The following user agent commands may be redefined for a particular object. For Menu objects, the default command is not redefinable:

Command	Object	Default Action
Copy	Menu object	No action (beep)
	Directory	Copy directory and contents
	Executable file	Copy the file
	File	Copy the file
Create	Menu object	No action (beep)
	Directory	Create empty directory
	Executable file	No action (beep)
	File	Create empty file
Delete	Menu object	No action (beep)
	Directory	Delete directory and contents
	Executable file	Delete file
	File	Delete file
Help	Menu object	No action (beep)
	Directory	Enter user agent help at directory node
	Executable file	Enter general user agent help
	File	Enter general user agent help
Move	Menu object	No action (beep)
	Directory	Move directory and contents
	Executable file	Move file
	File	Move file
Open	Menu object	No action (beep)
	Directory	Invoke files manager to display
	Executable file	Display file using default editor
	File	Display file
Print	Menu object	No action (beep)
	Directory	No action
	Executable file	Queue the file for printing
	File	Queue the file for printing

Rename	Menu object	No action (beep)
	Directory	Rename the directory
	Executable file	Rename the file
	File	Rename the file
Run	Menu object	No action (beep)
	Directory	No action (beep)
	Executable file	Create window and execute file
	File	No action (beep)

When no command is explicitly selected in the user agent, the default action varies as follows:

Menu object	Perform Open command
Directory	Perform Open command
Executable file	Perform Run command
File	Perform Open command

Except for Create, the above commands are obtained by selecting an object and then selecting either a command from the command menu, a function key, or function key label.

In the Create case, selecting Create yields a create menu, whether or not an object is selected. The initial Create menu contains File folder, Modem Profile, Phone Number, RS-232 Profile, and Standard file (ASCII file).

Suffixes

The **Suffixes** file defines objects in the class files. The minimum information which must be specified is the object Name, the identifying Suffix, a Description (for use in folder displays), and the default action.

The Name keyword definition begins the object definition. Subsequent keyword definitions are assumed to belong to this object until the next Name keyword is encountered. Other keywords that are typically defined (aside from the required set given above) are Create, Print, and Help.

The following keywords can be used in the **Suffixes** file:

Name = Object name

The object name is used in the create menu. If the Create keyword is not defined, the default action on create will be to create an empty file with the user specified name and the appropriate suffix. The object name should be relatively short, and embedded spaces are not allowed.

Suffix = file name suffix

The specified file name suffix is used to identify objects (files) of this type. For existing applications, two letter suffixes are used, a colon followed by an identifying letter. For example, *:W* for word processor documents, or *:S* for spreadsheets. The length of the user–supplied file name plus the suffix cannot exceed 14 characters.

Description = Object description

The object description is used in folder displays. It should

be short enough to fit on a line with the file name (say 50 characters max). If the first character of the description is an asterisk (*), then the description replaces the filename suffix in the folder display. Otherwise, the complete filename including the suffix is displayed.

Default = Command
Specifies which command is the default (typically Open or Run).

Open = Action specification
The action specification defines what to do when the file is opened via the user agent Open command. It typically involves creating a window and executing a process with the file passed to it as an argument. (See the "Action Specifications" section below for details.)

Keyword = Action specification
The remaining keywords that may be defined are all of this form. They are taken from the set of user agent commands listed in the section "Objects and Actions," and are optional. They should only be defined when the desired action differs from the default action.

Comm_pkgs
The **Comm_pkgs** file is used to support separately installed phone managers and terminal emulators, and to support external modems on any RS-232 port.

Each entry defines a communications application that is invoked through the phone manager. All applications that are invoked in this way must have an entry in the **Comm_pkgs** file. A single application may have multiple entries under different names, allowing several methods of invocation.

The following keywords can be used in the **Comm_pkgs** file:

Name =
Application package name.

Suffix=
The suffix the application package appends to its profiles.

Connection=
Defines the device type controlled by the application. It may be **ACU**, **DIR**, or **OBM**.

Create=
This is an optional field, depending upon the functionality supported by the application. It is the invocation sequence for the application to create a profile.

Modify=
This is also an optional field, depending upon the functionality supported by the application. It is the invocation sequence for the application to modify a profile.

Setup=
The standard invocation sequence for the package. The phone manager does the device "set-up" and dialing. The

child process inherits the file descriptors.

Nosetup=
The invocation sequence when no phone number is present in the data entry. No device "set-up" is provided.

Originate=
Specifies whether the application may be invoked from the console only, a remote terminal only, or from either. Allowed values are **CONSOLE**, **REMOTE**, or **BOTH**.

The "Setup" and "Nosetup" fields contain macros for substitution by the phone manager. The macros are limited to passing the following information to the application:

PHONENUM=phone number
DEVICE=full device path
PROFNAME=full path of profile entry
PID=process ID of the phone manager
HOSTNAME=phone directory name field
FN=phone device file descriptor

Check the table below to determine if the "Nosetup" or "Setup" action is desired.

Port and Device Type: Phone Manager Actions

Phone Number	Port Type	Device Type	Set-up	Phone Manager Dial	Invoke	Macro
yes	serial	ACU	yes	yes	yes	Setup
no	serial	ACU	no	no	yes	Nosetup
yes	serial	DIR	Error Condition			
no	serial	DIR	yes	no	yes	Setup
yes	ph	OBM	yes	yes	yes	Setup
no	ph	OBM	no	no	yes	Nosetup

The phone manager takes one of six paths. **Port Types** are either *serial* or *ph* (phone). **Device Types** are either an *ACU* (Automatic Calling Unit), which is a serial port to an external modem, *DIR* (direct) connection between a serial port and another computer, or *OBM* (On Board Modem) connecting to another computer. After determining whether or not to dial, the terminal emulator is invoked, and the macro Setup or Nosetup is used.

Menu **Objects**

The **Office, Administration,** and **Preferences** files define objects in the class menu objects. The minimum information which must be specified is the object Name and the action to take on Open.

The Name keyword definition begins the object definition. Subsequent keyword definitions are assumed to belong to this object, until the next Name keyword is encountered. The other keyword that is typically defined is Help.

The following keywords can be used in a menu objects file:

Name = Object name
: The object name is used when the menu is displayed. For example, the initial Office menu display consists of those names that are defined in the menu objects file **/usr/lib/ua/Office**. The object name should be relatively short.

Expert
: If the Expert keyword is present, then the menu item is only displayed in expert mode. For example, the UNIX object in the Office menu has this keyword in its definition.

MultiUser
: If the MultiUser keyword is present, then the menu item is only displayed in Multi-user mode. Multi-user mode can be changed via the user agent preferences form.

Default = Command
: Specifies which command is the default (typically Open or Run).

Open = Action specification
: The action specification defines what to do when the object is selected from the menu (or optionally when the object is opened via the user agent Open command). It typically involves creating a window and executing a process. (See the "Action Specifications" section below for details).

Keyword = Action specification
: The remaining keywords that may be defined are all of this form. They are taken from the set of user agent commands listed in the section "Objects and Actions," and are optional. They should only be defined when the desired action differs from the default action.

In general, Help is the only additional keyword defined. It is not desirable to allow the user to perform actions such as Copy, Create, Delete, Move, or Rename on menu objects. Those capabilities are generally provided via *install* and *remove* scripts. Experienced users, using the Bourne shell and a text editor, can create, delete, and rename objects at will.

Action Specifications

The action specification always starts with one of the following pseudo-commands:

Command	Arguments
UA	Menu objects file
FM	Directory
FO	Files
EXEC	File to execute
SH	File to execute as shell script
ERROR	String to display in message window

UA invokes another instance of the user agent, which takes as its menu the specified menu objects file.

FM invokes another instance of the files manager, which displays the specified directory in a new window (folder display).

FO performs a file operation on the specified files. The operation is specified via option characters as follows:

−c	Copy
−d	Delete (move to Wastebasket)
−m	Move
−r	Rename
−i	Destination is the invisible Clipboard

EXEC executes the specified file in a fork, and any subsequent arguments are interpreted and passed to the process as arguments.

SH executes the Bourne shell (**/bin/sh**) in a fork, and all arguments are interpreted and passed to the shell as arguments.

ERROR says that the command is illegal for the selected object. In the case of Create, *ERROR* prevents the object from being listed in the create menu.

EXEC and *SH* have a number of variations, which are used depending on the intelligence of the process being invoked. The basic *EXEC* and *SH* assume no intelligence. They create a window of the default size, and fix file descriptors 0, 1, and 2 to point to the window, then *exec* the process or the shell, and wait for its completion.

The variations are specified via option characters as follows:

−n	Run the process without a window
−w	Run the process without waiting
−d	Run the process in a dimensionless window
−p	Run the process with superuser privileges

The following interpretation is performed on all arguments to the commands:

%O is replaced with the list of currently selected objects.

%o is replaced with the name of the currently selected object. It is an error if a list of objects is specified by the user, and the action specification contains a **%o** argument.

%N and **%n** are the same as **%O** and **%o**, respectively, except that the suffixes are stripped off the filename.

***** is replaced with the list of files in the current directory, or in the specified directory.

Environment variables (which must start with $ a la the shell) are replaced with their current value.

In pathname specifications, the following substitutions are performed:

Filecabinet	$HOME/Filecabinet (or / if $HOME = /)
System	/
Floppydisk	/mnt
Wastebasket	$LOGDIR/Wastebasket
Clipboard	$LOGDIR/Clipboard
Parent	..

HOME and *LOGDIR* are the environment variables for the home directory and the login directory. They are the same in all cases except for the superuser, whose home directory is /, and whose login directory is the login directory of some selected user.

FM and *UA*, in spite of the description given above, don't actually create any new processes. In order to minimize the drain on system resources (primarily swap space), these commands merely create a new window with the appropriate display. The user agent knows about multiple windows, and responds to commands from any of the windows that it owns.

The windows created via UA and FM commands normally last until they are explicitly closed by the user. The **–e** option can be used on both these commands to create ephemeral windows. These windows are automatically closed after any action is taken on any of their objects. This option is useful in reducing "window clutter."

EXEC and *SH* (and all of their variations) require complete pathname specifications.

UA searches in the **/usr/lib/ua** directory for the menu objects file, and then in the login directory. If the file is found in both places, then the two files are merged to form a single menu.

In the case of an action specification for Create, **%o** refers to the name entered by the user in response to a prompt, and not to any preselected object.

Environment Variables

The user agent maintains the following environment variables and passes them on to all applications that it invokes:

SHELL	Path of default shell (initially **/bin/sh**)
EDIT	Path of default editor (initially **/bin/ed**)
LOGDIR	Path of login directory

In expert mode, the user can alter the environment variables *SHELL* and *EDIT*, using the user agent preferences form.

The user agent reads the values of its environment variables from the **Environment** files. The user agent Preferences form edits the **Environment** file in the login directory when a change is made.

SEE ALSO

uaupd(1), phone(7).

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

#include <sys/types.h>
#include <utmp.h>

DESCRIPTION

These files, which hold user and accounting information for such commands as *who*(1), *write*(1), and *login*(1M), have the following structure as defined by **<utmp.h>**:

```
#define    UTMP_FILE    "/etc/utmp"
#define    WTMP_FILE    "/etc/wtmp"
#define    ut_name      ut_user

struct utmp {
        char        ut_user[8];          /* User login name */
        char        ut_id[4];            /* /etc/inittab id */
                                         /* (usually line #) */
        char        ut_line[12];         /* device name */
                                         /* (console, lnxx) */
        short       ut_pid;              /* process id */
        short       ut_type;             /* type of entry */
        struct      exit_status {
           short       e_termination;    /* Process termination */
                                         /* status */
           short       e_exit;           /* Process exit status */
        } ut_exit;                       /* The exit status of a */
                                         /* process */
                                         /* marked as */
                                         /* DEAD_PROCESS. */
        time_t      ut_time;             /* time entry was made */
};

/* Definitions for ut_type */
#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5               /* Process spawned */
                                        /* by "init" */
#define LOGIN_PROCESS   6               /* A "getty" process */
                                        /* waiting for login */
#define USER_PROCESS    7               /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9
#define UTMAXTYPE       ACCOUNTING      /* Largest legal value */
                                        /* of ut_type */
```

```
/* Special strings or formats used in the "ut_line" field when */
/* accounting for something other than a process. */
/* No string for the ut_line field can be more than 11 chars + */
/* a NULL in length. */
#define RUNLVL_MSG "run-level %c"
#define BOOT_MSG   "system boot"
#define OTIME_MSG  "old time"
#define NTIME_MSG  "new time"
```

FILES

/usr/include/utmp.h
/etc/utmp
/etc/wtmp

SEE ALSO

login(1M), who(1), write(1), getut(3C).

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041 !  |042 "  |043 #  |044 $  |045 %  |046 &  |047 '  |
|050 (  |051 )  |052 *  |053 +  |054 ,  |055 -  |056 .  |057 /  |
|060 0  |061 1  |062 2  |063 3  |064 4  |065 5  |066 6  |067 7  |
|070 8  |071 9  |072 :  |073 ;  |074 <  |075 =  |076 >  |077 ?  |
|100 @  |101 A  |102 B  |103 C  |104 D  |105 E  |106 F  |107 G  |
|110 H  |111 I  |112 J  |113 K  |114 L  |115 M  |116 N  |117 O  |
|120 P  |121 Q  |122 R  |123 S  |124 T  |125 U  |126 V  |127 W  |
|130 X  |131 Y  |132 Z  |133 [  |134 \  |135 ]  |136 ^  |137 _  |
|140 `  |141 a  |142 b  |143 c  |144 d  |145 e  |146 f  |147 g  |
|150 h  |151 i  |152 j  |153 k  |154 l  |155 m  |156 n  |157 o  |
|160 p  |161 q  |162 r  |163 s  |164 t  |165 u  |166 v  |167 w  |
|170 x  |171 y  |172 z  |173 {  |174 |  |175 }  |176 ~  |177 del|

| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21 !  | 22 "  | 23 #  | 24 $  | 25 %  | 26 &  | 27 '  |
| 28 (  | 29 )  | 2a *  | 2b +  | 2c ,  | 2d -  | 2e .  | 2f /  |
| 30 0  | 31 1  | 32 2  | 33 3  | 34 4  | 35 5  | 36 6  | 37 7  |
| 38 8  | 39 9  | 3a :  | 3b ;  | 3c <  | 3d =  | 3e >  | 3f ?  |
| 40 @  | 41 A  | 42 B  | 43 C  | 44 D  | 45 E  | 46 F  | 47 G  |
| 48 H  | 49 I  | 4a J  | 4b K  | 4c L  | 4d M  | 4e N  | 4f O  |
| 50 P  | 51 Q  | 52 R  | 53 S  | 54 T  | 55 U  | 56 V  | 57 W  |
| 58 X  | 59 Y  | 5a Z  | 5b [  | 5c \  | 5d ]  | 5e ^  | 5f _  |
| 60 `  | 61 a  | 62 b  | 63 c  | 64 d  | 65 e  | 66 f  | 67 g  |
| 68 h  | 69 i  | 6a j  | 6b k  | 6c l  | 6d m  | 6e n  | 6f o  |
| 70 p  | 71 q  | 72 r  | 73 s  | 74 t  | 75 u  | 76 v  | 77 w  |
| 78 x  | 79 y  | 7a z  | 7b {  | 7c |  | 7d }  | 7e ~  | 7f del|
```

FILES

/usr/pub/ascii

NAME

environ – user environment

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*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (**:**). *Login*(1) sets **PATH=:/bin:/usr/bin**.

HOME Name of the user's login directory, set by *login*(1M) from the password file *passwd*(4).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1), which may exploit special capabilities of that terminal.

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

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

env(1), login(1M), sh(1), exec(2), getenv(3C), profile(4), term(5).

NAME

eqnchar – special character definitions for eqn and neqn

SYNOPSIS

eqn /usr/pub/eqnchar [files] | **troff** [options]

neqn /usr/pub/eqnchar [files] | **nroff** [options]

DESCRIPTION

Eqnchar contains *troff* and *nroff* character definitions for constructing characters that are not available on the Wang Laboratories, Inc. C/A/T phototypesetter. These definitions are primarily intended for use with *eqn*(1) and *neqn*; *eqnchar* contains definitions for the following characters:

ciplus	*ciplus*	| |	| |	*square*	*square*
citimes	*citimes*		*langle*	*langle*	*circlecircle*
wig	*wig*	*rangle*	*rangle*	*blot*	*blot*
−wig	*−wig*	*hbar*	*hbar*	*bullet*	*bullet*
>wig	*>wig*	*ppd*	*ppd*	*prop*	*prop*
<wig	*<wig*	*<->*	*<→*	*empty*	*empty*
=wig	*=wig*	*<=>*	*≤>*	*member*	*member*
star	*star*	| <	| <	*nomem*	*nomem*
bigstar	*bigstar*		| >	| >	*cupcup*
=dot	*=dot*	*ang*	*ang*	*cap*	*cap*
orsign	*orsign*	*rang*	*rang*	*incl*	*incl*
andsign	*andsign*		*3dot*	*3dot*	*subsetsubset*
=del	*=del*	*thf*	*thf*	*supset*	*supset*
oppA	*oppA*	*quarter*	*quarter*		*!subset!subset*
oppE	*oppE*	*3quarter*	*3quarter*		*!supset!supset*
angstrom	*angstrom*		*degree*	*degree*	*scrLscrL*
==<	*==<*	*==>*	*==>*		

FILES

/usr/pub/eqnchar

SEE ALSO

eqn(1), nroff(1).

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY  0
#define O_WRONLY  1
#define O_RDWR    2
#define O_NDELAY  04      /* Non-blocking I/O */
#define O_APPEND  010     /* append (writes guaranteed */
                          /* at the end) */
#define O_DIRECT  0100000/* Direct I/O */

/* Flag values accessible only to open(2) */
#define O_CREAT   00400 /* open with file create */
                        /* (uses third open arg)*/
#define O_TRUNC   01000 /* open with truncation */
#define O_EXCL    02000 /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD   0       /* Duplicate fildes */
#define F_GETFD   1       /* Get fildes flags */
#define F_SETFD   2       /* Set fildes flags */
#define F_GETFL   3       /* Get file flags */
#define F_SETFL   4       /* Set file flags */
#define F_GETLK   5       /* Get file lock */
#define F_SETLK   6       /* Set file lock */
#define F_SETLKW  7       /* Set file lock and wait */

/* File segment locking set data type - information passed
to system by user */

struct flock {
        short   l_type;
        short   l_whence;
        long    l_start;
        long    l_len;  /* len = 0 means until end of file */
        int     l_pid;
};

#define F_RDLCK   01
#define F_WRLCK   02
#define F_UNLCK   03
```

SEE ALSO

fcntl(2), open(2).

NAME

greek – graphics for the extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek [**|** **greek** –**T**terminal]

DESCRIPTION

Greek gives the mapping from ASCII to the "shift-out" graphics in effect between **SO** and **SI** on TELETYPE Model 37 terminals equipped with a 128-character type-box. These are the default greek characters produced by *nroff*. The filters of *greek*(1) attempt to print them on various other terminals. The file contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ϵ	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	$\neg$	_
partial	∂	]	integral	$\int$	^			

FILES

/usr/bin/greek

SEE ALSO

300(1), 4014(1), 450(1), greek(1), hp(1), tc(1), nroff(1).

NAME

man – macros for formatting entries in this manual

SYNOPSIS

nroff –man files

troff –man [**–rs1**] files

DESCRIPTION

These *troff* macros are used to lay out the format of the entries of this manual. A skeleton entry may be found in the file **/usr/man/u_man/man0/skeleton**. These macros are used by the *man*(1) command.

The default page size is $8.5'' \times 11''$, with a $6.5'' \times 10''$ text area; the **–rs1** option reduces these dimensions to $6'' \times 9''$ and $4.75'' \times 8.375''$, respectively; this option (which is *not* effective in *nroff*) also reduces the default type size from 10-point to 9-point, and the vertical line spacing from 12-point to 10-point. The **–rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters, the page length to 84 lines, and it inhibits underlining; this option should not be confused with the **–Tvp** option of the *man*(1) command, which is available at some UNIX sites.

Any *text* argument below may be one to six "words". Double quotes (*""*) may be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, **.I** may be used to italicize a whole line, or **.SM** followed by **.B** to make small bold text. By default, hyphenation is turned off for *nroff*, but remains on for *troff*.

Type font and size are reset to default values before each paragraph and after processing font- and size-setting macros, e.g., **.I**, **.RB**, **.SM**. Tab stops are neither used nor set by any macro except **.DT** and **.TH**.

Default units for indents *in* are ens. When *in* is omitted, the previous indent is used. This remembered indent is set to its default value (7.2 ens in *troff*, 5 ens in *nroff*–this corresponds to $0.5''$ in the default page size) by **.TH**, **.P**, and **.RS**, and restored by **.RE**.

.TH *t s c n* Set the title and entry heading; *t* is the title, *s* is the section number, *c* is extra commentary, e.g., "local", *n* is new manual name. Invokes **.DT** (see below).

.SH *text* Place subhead *text*, e.g., **SYNOPSIS**, here.

.SS *text* Place sub-subhead *text*, e.g., **Options**, here.

.B *text* Make *text* bold.

.I *text* Make *text* italic.

.SM *text* Make *text* 1 point smaller than default point size.

.RI *a b* Concatenate roman *a* with italic *b*, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold:

.IR .RB .BR .IB .BI

.P Begin a paragraph with normal font, point size, and indent. **.PP** is a synonym for **.P**.

.HP *in* Begin paragraph with hanging indent.

.TP *in* Begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. If the tag does not fit, it is printed on a separate line.

.IP *t in* Same as **.TP** *in* with tag *t*; often used to get an indented paragraph without a tag.

.RS *in* Increase relative indent (initially zero). Indent all output an extra *in* units from the current left margin.

.RE *k* Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level.

.PM *m* Produces proprietary markings; where *m* may be **P** for **PRIVATE**, **N** for **NOTICE**, **BP** for **BELL LABORATORIES PROPRIETARY**, or **BR** for **BELL LABORATORIES RESTRICTED**.

.DT Restore default tab settings (every 7.2 ens in *troff*, 5 ens in *nroff*).

.PD *v* Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (0.4v in *troff*, 1v in *nroff*).

The following *strings* are defined:

***R** in *troff*, **(Reg.)** in *nroff*.

***S** Change to default type size.

***(Tm** Trademark indicator.

The following *number registers* are given default values by **.TH**:

IN Left margin indent relative to subheads (default is 7.2 ens in *troff*, 5 ens in *nroff*).

LL Line length including **IN**.

PD Current interparagraph distance.

CAVEATS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *troff* and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

If a manual entry needs to be preprocessed by *cw*(1), *eqn*(1) (or *neqn*), and/or *tbl*(1), it must begin with a special line, causing the *man* command to invoke the appropriate preprocessor(s).

The programs that prepare the Table of Contents and the Permuted Index for this Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

name[, name, name . . .] \- explanatory text

The macro package increases the inter-word spaces (to eliminate ambiguity) in the *SYNOPSIS* section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font by the constant-width font–see *cw*(1)). Of course, if the input text of an entry contains requests for other fonts (e.g., **.I**, **.RB**, \fI), the corresponding fonts must be mounted.

FILES

/usr/lib/tmac/tmac.an
/usr/lib/macros/cmp.[nt].[dt].an
/usr/lib/macros/ucmp.[nt].an
/usr/man/[ua]_man/man0/skeleton

SEE ALSO

nroff(1).

BUGS

If the argument to **.TH** contains *any* blanks and is *not* enclosed by double quotes (*""*), there will be bird-dropping-like things on the output.

NAME

mm – the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff **–mm** [options] [files]

nroff **–cm** [options] [files]

mmt [options] [files]

troff **–mm** [options] [files]

troff **–cm** [options] [files]

DESCRIPTION

This package provides a formatting capability for a very wide variety of documents. It is the standard package used by the BTL typing pools and documentation centers. The manner in which a document is typed in and edited is essentially independent of whether the document is to be eventually formatted at a terminal or is to be phototypeset. See the references below for further details.

The **–mm** option causes *nroff* and *troff* to use the non-compacted version of the macro package, while the **–cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/mm[nt]	non-compacted version of the package
/usr/lib/macros/cmp.[nt].[dt].m	compacted version of the package
/usr/lib/macros/ucmp.[nt].m	initializers for the compacted version of the package

SEE ALSO

mm(1), nroff(1).

MM–Memorandum Macros by D. W. Smith and J. R. Mashey.

Typing Documents with MM by D. W. Smith and E. M. Piskorik.

NAME

modemcap – modem capability data base

SYNOPSIS

/usr/lib/uucp/modemcap

DESCRIPTION

Modemcap is a data base describing modems, in the same manner as termcap describes terminals. Modems described in the modemcap data base and connected to an RS-232 port can then be passed commands by standard *dial*(3) routines.

Modems named in modemcap are identified in the **L-devices** file, which is maintained by the Administration software, as follows:

ACU tty000 name speed

ACU defines the modem to *uucp*, **tty000** is the device used, **name** is the name given to the modem in the modemcap entry, and **speed** is the baud rate used by the modem.

See *termcap*(5) for a description of the format used for modemcap entries.

Commands

Where termcap defines terminal capabilities, modemcap defines modem commands. The available commands are described in the table below. See *termcap*(5) for an explanation of the command types and syntax.

Name	Type	Description
a[a–z,0–9]	str	Abort EQUAL with string as error
b[a–z,0–9]	str	Abort NOT_EQUAL with string as error
c[a–z,0–9]	str	Compare string to previous results of 'w' (not including terminator)
d[a–z,0–9]	num	Delay num seconds
eh	str	End of phone string
es	char	primary Command start character
m[a–z,0–9]	num	Skip num instructions EQUAL
n[a–z,0–9]	num	Skip num instructions NOT_EQUAL
pa	char	Pause character (replaces [–])
ph	str	Send (srt,phone#,eh). If not defined, then no string sent
pp	str	Controls modem for pulse dialing
ps	char	Primary command start character
pt	str	Controls modem for tone dialing
pw	char	Wait character (replaces [w=])
s[a–z,0–9]	str	Send (ps,str,es) if ps and es are defined
t[a–z,0–9]	str	Send str
w[a–z,0–9]	char	Read characters until get character specified (nulls are ignored)

FILES

/usr/lib/uucp/modemcap	file containing modem descriptions
/usr/lib/uucp/L–devices	logical device identification file

SEE ALSO

termcap(5), dial(3), uucp(1).

NAME

mptx – the macro package for formatting a permuted index

SYNOPSIS

nroff –mptx [options] [files]

DESCRIPTION

This package provides a definition for the **.xx** macro used for formatting a permuted index as produced by *ptx*(1). This package does not provide any other formatting capabilities such as headers and footers. If these or other capabilities are required, the *mptx* macro package may be used in conjunction with the *MM* macro package. In this case, the **–mptx** option must be invoked *after* the **–mm** call. For example:

```
nroff –cm –mptx file
```

or

```
mm –mptx file
```

FILES

/usr/lib/tmac/tmac.ptx	pointer to the non-compacted version of the package
/usr/lib/macros/ptx	non-compacted version of the package

SEE ALSO

mm(1), nroff(1), ptx(1), mm(5).

NAME

regexp – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;

int step(string, expbuf)
char *string, *expbuf;
```

DESCRIPTION

This page describes general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *expr*(1), etc., which perform regular expression matching, use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

ERROR(*val*) This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

compile(instring, expbuf, endbuf, eof)

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf*–*expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

step(string, expbuf)

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns a one indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called, simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at some time during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used be *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT        register char *sp = instring;
#define GETC( )     (*sp++)
#define PEEKC( )    (*sp)
```

```
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr( )
#include <regexp.h>
...
        compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
        if(step(linebuf, expbuf))
                succeed( );
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The handling of *circf* is kludgy.
The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.
The actual code is probably easier to understand than this manual page.

NAME

stat – data returned by stat system call

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
/*
 * Structure of the result of stat
 */

struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};

#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFIFO  0010000 /* fifo */
#define S_ISUID  04000   /* set user id on execution */
#define S_ISGID  02000   /* set group id on execution */
#define S_ISVTX  01000   /* save swapped text even after */
                         /* use */
#define S_IREAD  00400   /* read permission, owner */
#define S_IWRITE 00200   /* write permission, owner */
#define S_IEXEC  00100   /* execute/search permission, */
                         /* owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), types(5).

Commands whose behavior depends on the type of terminal should accept arguments of the form **-T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **$TERM**, which, in turn, should contain *term*.

SEE ALSO

mm(1), nroff(1), sh(1), stty(1), tabs(1), profile(4), environ(5).

BUGS

This is a small candle trying to illuminate a large, dark problem. Programs that ought to adhere to this nomenclature do so somewhat fitfully.

NAME

term – conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *nroff*, *mm*(1), *tabs*(1)) and are maintained as part of the shell environment (see *sh*(1), *profile*(4), and *environ*(5)) in the variable **$TERM**:

1520	Datamedia 1520
1620	Diablo 1620 and others using the HyType II printer
1620–12	same, in 12-pitch mode
2621	Hewlett-Packard HP2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer – compressed mode
2631–e	Hewlett-Packard 2631 line printer – expanded mode
2640	Hewlett-Packard HP2640 series
2645	Hewlett-Packard HP264n series (other than the 2640 series)
300	DASI/DTC/GSI 300 and others using the HyType I printer
300–12	same, in 12-pitch mode
300s	DASI/DTC/GSI 300s
382	DTC 382
300s–12	same, in 12-pitch mode
3045	Datamedia 3045
33	TELETYPE Model 33 KSR
37	TELETYPE Model 37 KSR
40–2	TELETYPE Model 40/2
40–4	TELETYPE Model 40/4
4540	TELETYPE Model 4540
3270	IBM Model 3270
4000a	Trendata 4000a
4014	Tektronix 4014
43	TELETYPE Model 43 KSR
450	DASI 450 (same as Diablo 1620)
450–12	same, in 12-pitch mode
735	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
sync	generic name for synchronous TELETYPE 4540-compatible terminals
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300

Up to 8 characters, chosen from [– a–z 0–9], make up a basic terminal name. Terminal sub-models and operational modes are distinguished by suffixes beginning with a –. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, e.g., by *vi*(1). 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 the number of 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 "
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 "
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
10-19	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
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)

Additional capabilities used by *tam*(3T):

Name	Type	Pad?	Description
BE	str		Bold end
BO	str		Bold on
CI	str		Cursor invisible
CV	str		Cursor visible
DE	str		Dim end
DS	str		Dim start
EE	str		End every attribute
FE	str		Turn off SLK labels
FL	str		Set SLK label (printf fmt string)
KM	str		input key map (full pathname)
XE	str		Overstrike end
XS	str		Overstrike start

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 used as an example only.)

```
C1|c100|concept100:is=\EU\Ef\E7\E5\E8\E1\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*:l24:mi:nd=\E=:\
:se=\Ed\Ee:so=\ED\EE:a=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 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 delays, numeric capabilities, 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** to 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** and **\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 **\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 building 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 test a new terminal description you can set the environment variable TERMCAP to a path name 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, 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
%D	Reverse coding (x-2

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 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 you 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 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, this is acceptable, and although it may confuse some programs slightly, it can't be helped.

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).

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* two-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 *xx*.

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 characters. 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 cancelled 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.

TAM Capabilities

The additional capabilities provided for use with *tam*(3T) are all caps to distinguish them from the standard capabilities. **EE** tells TAM that the terminal uses ANSI-style character attributes, i.e., the strings that turn attributes on are cumulative and there is one string, **EE**, that turns all attributes off. If the **EE** capability is included in the *termcap* entry, none of the other attribute end strings, **BE**, **XE**, and **DE**, need be defined.

A terminal with **sg** set is treated as if it has no attributes.

On a terminal with **us** but *not* **so** defined, **so** is set to **us**.

TAM uses attributes to show selected and unselected windows, and menu and form cursors. On a terminal with no attributes, the border of a selected window is drawn with "*" and the border of an unselected window is drawn with ".". On a terminal with

attributes, the attributes used to draw window borders depend on the **so**, **BO**, and **DS** capabilities. If only **so** is defined, a selected window border is drawn using spaces with the **so** attribute, and unselected window borders are drawn with ".". If **so** and **BO** are defined, a selected window border is drawn using spaces with the **BO** attribute, and unselected window borders are drawn using spaces with the **so** attribute. If **so** and **DS** are defined, a selected window border is drawn using spaces with the **so** attribute, and unselected window borders are drawn using spaces with the **DS** attribute.

FE and **FL** are used for terminals that have hardware SLK labels, such as the b513. **FL** is a *printf* format string requiring two arguments: the key number and the label string. **FE** turns off the SLK labels.

KM is the full pathname of the file TAM uses to translate keyboard input sequences into their UNIX PC equivalent. By convention these mapping files are named **kmap.**<*terminal-name*> and are located in **/usr/lib/ua**.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), tset(1), vi(1), more(1), tam(3T).

BUGS

Ex allows only 256 characters for string capabilities. 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

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
typedef  struct { int r[1]; } *        physadr;
typedef  long                  daddr_t;
typedef  char *                caddr_t;
typedef  unsigned int          uint;
typedef  unsigned short        ushort;
typedef  ushort                ino_t;
typedef  short                 cnt_t;
typedef  long                  time_t;
typedef  int                   label_t[10];
typedef  short                 dev_t;
typedef  long                  off_t;
typedef  long                  paddr_t;
typedef  long                  key_t;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(4). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(4).

NAME

varargs – handle variable argument list

SYNOPSIS

```
#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type va_arg(pvar, type)
va_list pvar;

void va_end(pvar)
va_list pvar;
```

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists [such as *printf*(3S)] but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

The **va_alist** is used as the parameter list in a function header.

The **va_dcl** is a declaration for *va_alist*. No semicolon should follow *va_dcl*.

The **va_list** is a type defined for the variable used to traverse the list.

The **va_start** is called to initialize *pvar* to the beginning of the list.

The **va_arg** will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

The **va_end** is used to clean up.

Multiple traversals, each bracketed by *va_start* ... *va_end*, are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#include <varargs.h>
#define MAXARGS   100

/*      execl is called by */
/*              execl(file, arg1, arg2, ..., (char*)0); */

execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
```

```
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *))
        != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

SEE ALSO

exec(2), printf(3S), vprintf(3S).

NOTES

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.

It is non-portable to specify a second argument of *char*, *short*, or *float* to *va_arg*, since arguments seen by the called function are not *char*, *short*, or *float*. C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.