AT&T

999-801-312IS
For use with 3.51 Software

AT&T UNIX® PC

UNIX System V User's Manual Volume II

NOTICE

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.

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.
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.
login(1)	This command is not available on the UNIX PC for Version 3.5, but is available on

	System V Release 2.
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.
getenv(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.
strcmp(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. System Maintenance Commands

7. Special Files

NAME

intro – introduction to maintenance and application programs

DESCRIPTION

This section describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes. The commands in this section should be used along with those listed in Section 1 of the *UNIX System User's Manual*. References to other manual entries not of the form *name*(1M) or *name*(7) refer to entries of that manual.

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).
UNIX System User's Manual.
UNIX System Administrator's Guide.

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

accept, reject – allow/prevent LP requests

SYNOPSIS

/usr/lib/accept destinations
/usr/lib/reject [**−r**[reason]] destinations

DESCRIPTION

Accept allows *lp*(1) to accept requests for the named *destinations*. A *destination* can be either a printer or a class of printers. Use *lpstat*(1) to find the status of *destinations*.

Reject prevents *lp*(1) from accepting requests for the named *destinations*. A *destination* can be either a printer or a class of printers. Use *lpstat*(1) to find the status of *destinations*. The following option is useful with *reject*.

−r[*reason*] Associates a *reason* with preventing *lp* from accepting requests. This *reason* applies to all printers mentioned up to the next **−r** option. *Reason* is reported by *lp* when users direct requests to the named *destinations* and by *lpstat*(1). If the **−r** option is not present or the **−r** option is given without a *reason*, then a default *reason* will be used.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1), lpadmin(1M), lpsched(1M), lpstat(1).

NAME

bcopy – interactive block copy

SYNOPSIS

/etc/bcopy

DESCRIPTION

Bcopy dates from a time when neither the UNIX file system nor the DEC disk drives were as reliable as they are now. *Bcopy* copies from and to files starting at arbitrary block (512-byte) boundaries.

The following questions are asked:

to: (you name the file or device to be copied to).
offset: (you provide the starting "to" block number).
from: (you name the file or device to be copied from).
offset: (you provide the starting "from" block number).
count: (you reply with the number of blocks to be copied).

After **count** is exhausted, the **from** question is repeated (giving you a chance to concatenate blocks at the **to+offset+count** location). If you answer **from** with a carriage return, everything starts over.

Two consecutive carriage returns terminate *bcopy*.

SEE ALSO

cpio(1), dd(1).

NAME

chroot – change root directory for a command

SYNOPSIS

/etc/chroot newroot command

DESCRIPTION

The given command is executed *relative to the new root*. The meaning of any initial slashes (/) in path names is changed for a command and any of its children to *newroot*. Furthermore, the initial working directory is *newroot*.

Notice that:

chroot newroot command >x

will create the file **x** relative to the original root, not the new one.

This command is restricted to the super-user.

The new root path name is always relative to the current root: even if a *chroot* is currently in effect, the *newroot* argument is relative to the current root of the running process.

SEE ALSO

chdir(2).

BUGS

One should exercise extreme caution when referencing special files in the new root file system.

NAME

clri – clear i-node

SYNOPSIS

/etc/clri file-system i-number ...

DESCRIPTION

Clri writes zeros on the 64 bytes occupied by the i-node numbered *i-number*. *File-system* must be a special file name referring to a device containing a file system. After *clri* is executed, any blocks in the affected file will show up as "missing" in an *fsck*(1M) of the *file-system*. This command should only be used in emergencies and extreme care should be exercised.

Read and write permission is required on the specified *file-system* device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to *zap* an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

SEE ALSO

fsck(1M), fsdb(1M), ncheck(1M), fs(4).

BUGS

If the file is open, *clri* is likely to be ineffective.

NAME

cron, smgr – clock daemon

SYNOPSIS

/etc/cron
/etc/smgr

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file **/usr/lib/crontab**. Because *cron* never exits, it should be executed only once. This is best done by running *cron* from the initialization process through the file **/etc/rc** (see *init*(1M)).

In the UNIX PC, the status manager (**/etc/smgr**), which displays the date, time, and message icons on the screen, includes the functionality of *cron*. Thus *cron* is not run on the UNIX PC if the status manager is used.

The file **crontab** consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify in order:

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
and day of the week (0-6, with 0=Sunday).

Each of these patterns may contain:

a number in the (respective) range indicated above;
two numbers separated by a minus (indicating an inclusive range);
a list of numbers separated by commas (meaning all of these numbers); or
an asterisk (meaning all legal values).

The sixth field is a string that is executed by the shell at the specified time(s). A % in this field is translated into a new-line character. Only the first line (up to a % or the end of line) of the command field is executed by the shell. The other lines are made available to the command as standard input.

Cron examines **crontab** once a minute to see if it has changed; if it has, *cron* reads it. Thus it takes only a minute for entries to become effective.

FILES

/usr/lib/crontab
/usr/adm/cronlog

SEE ALSO

init(1M), sh(1).

DIAGNOSTICS

A history of all actions by *cron* are recorded in **/usr/adm/cronlog**.

BUGS

Cron reads **crontab** only when it has changed, but it reads the in-core version of that table once a minute. A more efficient algorithm could be used. The overhead in running *cron* is about one percent of the CPU, exclusive of any commands executed by *cron*.

NAME

devnm – device name

SYNOPSIS

/etc/devnm [names]

DESCRIPTION

Devnm identifies the special file associated with the mounted file system where the argument *name* resides (as a special case, both the block device name and the swap device name is printed for the argument name / if swapping is done on the same disk section as the **root** file system). Argument names must be full path names.

This command is most commonly used by **/etc/rc** (see *rc*(1M)) to construct a mount table entry for the **root** device.

EXAMPLE

The command:

```
/etc/devnm   /
```

produces

```
fp002   /
```

if /dev/fp002 is mounted on **/**.

Or the command:

```
/etc/devnm   /u
```

produces

```
fp003   /u
```

if /dev/fp003 is mounted on /u.

FILES

/etc/mnttab

SEE ALSO

rc(1M), setmnt(1M).

NAME

df – report number of free disk blocks

SYNOPSIS

df [**−t**] [**−f**] [file-systems]

DESCRIPTION

Df prints out the number of free blocks and free i-nodes available for on-line file systems by examining the counts kept in the super-blocks; *file-systems* may be specified either by device name (e.g., **/dev/fp002**) or by mounted directory name (e.g., **/usr**). If the *file-systems* argument is unspecified, the free space on all of the mounted file systems is printed.

The **−t** flag causes the total allocated block figures to be reported as well.

If the **−f** flag is given, only an actual count of the blocks in the free list is made (free i-nodes are not reported). With this option, *df* will report on raw devices.

FILES

/dev/fp*
/etc/mnttab

SEE ALSO

fs(4), mnttab(4).

NAME

dismount – remove floppy or cartridge disk

SYNOPSIS

dismount [**–f**] [**–s**]

DESCRIPTION

DISMOUNT prevents damage to a floppy or cartridge disk caused by sudden removal of the disk from its drive. The program waits for pending input/output on the disk to complete, forbids further input/output to the disk, unmounts the disk's file systems, and clears the pulled flag in the disk's volume home block. When *dismount* finishes without error, it is safe to take the disk out of the drive.

–f is the default and dismounts the floppy disk. **–s** is historical.

A disk that was removed without a prior dismount is noticeable because its pulled flag is still set. Inserting such a disk in the drive causes UNIX to print a warning on the system console. If you receive such a warning, check the consistency of file systems and databases on the disk.

FILES

/etc/mnttab - mounted file system list

SEE ALSO

fsck(1M), update(1), gd(7).

NAME

fsck – file system consistency check and interactive repair

SYNOPSIS

/etc/fsck [–**y**] [–**n**] [–**s**X] [–**S**X] [–**t** file] [–**q**] [–**D**] [–**f**] [–**p**] [file-systems]

DESCRIPTION

Fsck

Fsck audits and interactively repairs inconsistent conditions for UNIX file systems. If the file system is consistent then the number of files, number of blocks used, and number of blocks free are reported. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. It should be noted that most corrective actions will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission *fsck* will default to a –**n** action.

Fsck has more consistency checks than its predecessors *check*, *dcheck*, *fcheck*, and *icheck* combined.

The following options are interpreted by *fsck*.

–**y** Assume a yes response to all questions asked by *fsck*.

–**n** Assume a no response to all questions asked by *fsck*; do not open the file system for writing.

–**s***X* Ignore the actual free list and (unconditionally) reconstruct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done; if this is not possible, care should be taken that the system is quiescent and that it is rebooted immediately afterwards. This precaution is necessary so that the old, bad, in-core copy of the superblock will not continue to be used, or written on the file system.

The –**s***X* option allows for creating an optimal free-list organization. The following forms of *X* are supported for the following devices:

–s3 (RP03)
–s4 (RP04, RP05, RP06)
–sBlocks-per-cylinder:Blocks-to-skip
(for anything else)

If *X* is not given, the values used when the file system was created are used. If these values were not specified, then the value *400:7* is used.

–S*X* Conditionally reconstruct the free list. This option is like –**s***X* above except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using –S will force a no response to all questions asked by *fsck*. This option is useful for forcing free list reorganization on uncontaminated file systems.

−t If *fsck* cannot obtain enough memory to keep its tables, it uses a scratch file. If the **−t** option is specified, the file named in the next argument is used as the scratch file, if needed. Without the **−t** flag, *fsck* will prompt the operator for the name of the scratch file. The file chosen should not be on the file system being checked, and if it is not a special file or did not already exist, it is removed when *fsck* completes.

−q Quiet *fsck*. Do not print size-check messages in Phase 1. Unreferenced **fifos** will silently be removed. If *fsck* requires it, counts in the superblock will be automatically fixed and the free list salvaged.

−D Directories are checked for bad blocks. Useful after system crashes.

−f Fast check. Check block and sizes (Phase 1) and check the free list (Phase 5). The free list will be reconstructed (Phase 6) if it is necessary.

−p Preen file systems only. Assume that no operator is present: fix minor problems without asking permission and if there are major problems, note them and exit with an error status. Only the following problems are considered minor:

Unreferenced inodes.
Link counts in inodes too large.
Missing blocks in the free list.
Blocks in the free list also in files.
Counts in the super block wrong.

The **−p** option allows a normal boot without operator intervention. The startup script that runs *fsck* (**/etc/rc** on the UNIX PC) can specify the **−p** option to *fsck* and make a normal boot contingent upon a normal *fsck* return status.

If no *file-systems* are specified, *fsck* will read a list of default file systems from the file **/etc/checklist**.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Incorrect number of blocks.
   Directory size not 16-byte aligned.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.

8. Super Block checks:
   More than 65536 inodes.
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the **lost+found** directory, if the files are nonempty. The user will be notified if the file or directory is empty or not. If it is empty, *fsck* will silently remove them. *Fsck* will force the reconnection of nonempty directories. The name assigned is the inode number. The only restriction is that the directory **lost+found** must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This is accomplished by making **lost+found**, copying a number of files to the directory, and then removing them (before *fsck* is executed).

Checking the raw device is almost always faster and should be used with everything but the *root* file system.

FILES

/etc/checklist	contains default list of file systems to check.

SEE ALSO

clri(1M), ncheck(1M), checklist(4), fs(4).
Setting up UNIX in the *UNIX System Administrator's Guide*.

BUGS

Inode numbers for **.** and **..** in each directory should be checked for validity.

DIAGNOSTICS

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

If **−p** was specified and preening was inadequate, a nonzero status is returned.

NAME

fsdb – file system debugger

SYNOPSIS

/etc/fsdb special [–]

DESCRIPTION

Fsdb can be used to patch up a damaged file system after a crash. It has conversions to translate block and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an i-node. These greatly simplify the process of correcting control block entries or descending the file system tree.

Fsdb contains several error checking routines to verify i-node and block addresses. These can be disabled if necessary by invoking *fsdb* with the optional – argument or by the use of the **O** symbol. (*Fsdb* reads the i-size and f-size entries from the superblock of the file system as the basis for these checks.)

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

Fsdb reads a block at a time and will therefore work with raw as well as block I/O. A buffer management routine is used to retain commonly used blocks of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding block.

The symbols recognized by *fsdb* are:

#	absolute address
i	convert from i-number to i-node address
b	convert to block address
d	directory slot offset
+,–	address arithmetic
q	quit
>,<	save, restore an address
=	numerical assignment
=+	incremental assignment
=–	decremental assignment
="	character string assignment
O	error checking flip flop
p	general print facilities
f	file print facility
B	byte mode
W	word mode
D	double word mode
!	escape to shell

The print facilities generate a formatted output in various styles. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the delete character. If a number follows the **p** symbol, that many entries are printed. A check is made to detect block boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current block are printed. The print options available are:

i	print as i-nodes
d	print as directories
o	print as octal words
e	print as decimal words
c	print as characters
b	print as octal bytes

The **f** symbol is used to print data blocks associated with the current i-node. If followed by a number, that block of the file is printed. (Blocks are numbered from zero.) The desired print option letter follows the block number, if present, or the **f** symbol. This print facility works for small as well as large files. It checks for special devices and that the block pointers used to find the data are not zero.

Dots, tabs and spaces may be used as function delimiters but are not necessary. A line with just a new-line character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or i-node, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal address followed by the value in octal and decimal. A **.B** or **.D** is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. Inodes are printed with labeled fields describing each element.

The following mnemonics are used for i-node examination and refer to the current working i-node:

md	mode
ln	link count
uid	user ID number
gid	group ID number
sz	file size
a#	data block numbers (0 – 12)
at	access time
mt	modification time
maj	major device number
min	minor device number

EXAMPLES

386i	prints i-number 386 in an i-node format. This now becomes the current working i-node.
ln=4	changes the link count for the working i-node to 4.
ln=+1	increments the link count by 1.
fc	prints, in ASCII, block zero of the file associated with the working i-node.

2i.fd	prints the first 32 directory entries for the root i-node of this file system.
d5i.fc	changes the current i-node to that associated with the 5th directory entry (numbered from zero) found from the above command. The first logical block of the file is then printed in ASCII.
512B.p0o	prints the superblock of this file system in octal.
2i.a0b.d7=3	changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line.
d7.nm="name"	changes the name field in the directory slot to the given string. Quotes are optional when used with **nm** if the first character is alphabetic.
a2b.p0d	prints the third block of the current inode as directory entries.

SEE ALSO

fsck(1M), dir(4), fs(4).

NAME

fuser – identify processes using a file or file structure

SYNOPSIS

/etc/fuser [**–ku**] files [–] [[**–ku**] files]

DESCRIPTION

Fuser lists the process IDs of the processes using the *files* specified as arguments. For block special devices, all processes using any file on that device are listed. The process ID is followed by **c**, **p** or **r** if the process is using the file as its **c**urrent directory, the **p**arent of its current directory (only when in use by the system), or its **r**oot directory, respectively. If the **–u** option is specified, the login name, in parentheses, also follows the process ID. In addition, if the **–k** option is specified, the **SIGKILL** signal is sent to each process. Only the super-user can terminate another user's process (see *kill*(2)). Options may be respecified between groups of files. The new set of options replaces the old set, with a lone dash canceling any options currently in force.

The process IDs are printed as a single line on the standard output, separated by spaces and terminated with a single new line. All other output is written on standard error.

EXAMPLES

fuser –ku /dev/dsk1?

> will terminate all processes that are preventing disk drive one from being unmounted if typed by the super-user, listing the process ID and login name of each as it is killed.

fuser –u /etc/passwd

> will list process IDs and login names of processes that have the password file open.

fuser –ku /dev/dsk1? –u /etc/passwd

> will do both of the above examples in a single command line.

Note that the above device names for disks are generic to the 3B20S and may be different on other processors.

FILES

/unix	for namelist
/dev/kmem	for system image
/dev/mem	also for system image

SEE ALSO

mount(1M), ps(1), kill(2), signal(2).

NAME

getty – set terminal type, modes, speed, and line discipline

SYNOPSIS

/etc/getty [**–h**] [**–t** timeout] line [speed [type [linedisc]]]

/etc/getty –c file

DESCRIPTION

Getty is a program that is invoked by *init*(1M). It is the second process in the series, (*init-getty-login-shell*) that ultimately connects a user with UNIX. Initially *getty* generates a system identification message from the values returned by the *uname*(2) system call. Then, if */etc/issue* exists, it outputs this to the user's terminal, followed finally by the login message field for the entry it is using from **/etc/gettydefs**. *Getty* reads the user's login name and invokes the *login*(1M) command with the user's name as argument. While reading the name, *getty* attempts to adapt the system to the speed and type of terminal being used.

Line is the name of a tty line in **/dev** to which *getty* is to attach itself. *Getty* uses this string as the name of a file in the **/dev** directory to open for reading and writing. Unless *getty* is invoked with the **–h** flag, *getty* will force a hangup on the line by setting the speed to zero before setting the speed to the default or specified speed. The **–t** flag plus *timeout* in seconds, specifies that *getty* should exit if the open on the line succeeds and no one types anything in the specified number of seconds. The optional second argument, *speed,* is a label to a speed and tty definition in the file **/etc/gettydefs**. This definition tells *getty* what speed to initially run at, what the login message should look like, what the inital tty settings are, and what speed to try next should the user indicate that the speed is inappropriate. (By typing a <*break*> character.) The default *speed* is 300 baud. The optional third argument, *type*, is a character string describing to *getty* what type of terminal is connected to the line in question. *Getty* understands the following types:

none	default
vt61	DEC vt61
vt100	DEC vt100
hp45	Hewlett-Packard HP45
c100	Concept 100

The default terminal is **none**; i.e., any crt or normal terminal unknown to the system. Also, for terminal type to have any meaning, the virtual terminal handlers must be compiled into the operating system. They are available, but not compiled in the default condition. The optional fourth argument, *linedisc*, is a character string describing which line discipline to use in communicating with the terminal. Again the hooks for line disciplines are available in the operating system but there is only one presently available, the default line discipline, **LDISC0**.

When given no optional arguments, *getty* sets the *speed* of the interface to 300 baud, specifies that raw mode is to be used (awaken on every character), that echo is to be suppressed, either parity allowed, newline characters will be converted to carriage return-line feed, and tab expansion performed on the standard output. It types the login message before reading the user's name a character at a time. If a null character (or framing error) is received, it is assumed to be the result of the user pushing the "break" key. This will cause *getty* to attempt the next *speed* in the series. The series that *getty* tries is determined by what it finds in **/etc/gettydefs**.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *ioctl*(2)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

In addition to the standard UNIX erase and kill characters (# and @), *getty* also understands **\b** and **^U**. If the user uses a **\b** as an erase, or **^U** as a kill character, *getty* sets the standard erase character and/or kill character to match.

Getty also understands the "standard" ESS2 protocols for erasing, killing and aborting a line, and terminating a line. If *getty* sees the ESS erase character, _, or kill character, $, or abort character, &, or the ESS line terminators, / or !, it arranges for this set of characters to be used for these functions.

Finally, *login* is called with the user's name as an argument. Additional arguments may be typed after the login name. These are passed to *login*, which will place them in the environment (see *login*(1M)).

A check option is provided. When *getty* is invoked with the **−c** option and *file*, it scans the file as if it were scanning **/etc/gettydefs** and prints out the results to the standard output. If there are any unrecognized modes or improperly constructed entries, it reports these. If the entries are correct, it prints out the values of the various flags. See *ioctl*(2) to interpret the values. Note that some values are added to the flags automatically.

FILES

/etc/gettydefs
/etc/issue

SEE ALSO

init(1M), login(1M), ioctl(2), gettydefs(4), inittab(4), tty(7).

BUGS

While *getty* does understand simple single character quoting conventions, it is not possible to quote the special control characters that *getty* uses to determine when the end of the line has been reached, which protocol is being used, and what the erase character is. Therefore it is not possible to login via *getty* and type a #, **@**, /, !, _, backspace, **^U**, **^D**, or & as part of your login name or arguments. They will always be interpreted as having their special meaning as described above.

NAME

init, telinit – process control initialization

SYNOPSIS

/etc/init [**0123456SsQq**]

/etc/telinit [**0123456sSQqabc**]

DESCRIPTION

Init

Init is a general process spawner. Its primary role is to create processes from a script stored in the file **/etc/inittab** (see *inittab*(4)). This file usually has *init* spawn *getty*'s on each line that a user may log in on. It also controls autonomous processes required by any particular system.

Init considers the system to be in a *run-level* at any given time. A *run-level* can be viewed as a software configuration of the system where each configuration allows only a selected group of processes to exist. The processes spawned by *init* for each of these *run-levels* is defined in the *inittab* file. *Init* can be in one of eight *run-levels*, **0–6** and **S** or **s**. The *run-level* is changed by having a privileged user run **/etc/init** (which is linked to */etc/telinit*). This user spawned *init* sends appropriate signals to the original *init* spawned by the operating system when the system was rebooted, telling it which *run-level* to change to.

Init is invoked inside UNIX as the last step in the boot procedure. The first thing *init* does is to look for **/etc/inittab** and see if there is an entry of the type *initdefault* (see *inittab*(4)). If there is, *init* uses the *run-level* specified in that entry as the initial *run-level* to enter. If this entry is not in *inittab* or *inittab* is not found, *init* requests that the user enter a *run-level* from the virtual system console, **/dev/syscon.** If an **S** (**s**) is entered, *init* goes into the *SINGLE USER* level. This is the only *run-level* that doesn't require the existence of a properly formated *inittab* file. If **/etc/inittab** doesn't exist, then by default the only legal *run-level* that *init* can enter is the *SINGLE USER* level. In the *SINGLE USER* level the virtual console terminal **/dev/syscon** is opened for reading and writing and the command **/bin/su** is invoked immediately. To exit from the *SINGLE USER run-level* one of two options can be elected. First, if the shell is terminated (via an end-of-file), *init* will reprompt for a new *run-level*. Second, the *init* or *telinit* command can signal *init* and force it to change the *run-level* of the system.

When attempting to boot the system, failure of *init* to prompt for a new *run-level* may be due to the fact that the device **/dev/syscon** is linked to a device other than the physical system teletype (**/dev/systty**). If this occurs, *init* can be forced to relink **/dev/syscon** by typing a delete on the system teletype which is co-located with the processor.

When *init* prompts for the new *run-level*, the operator may only enter one of the digits **0** through **6** or the letters **S** or **s**. If **S** is entered *init* operates as previously described in *SINGLE USER* mode with the additional result that **/dev/syscon** is linked to the user's terminal line, thus making it the virtual system console. A message is generated on the physical console, **/dev/systty**, saying where the virtual terminal has been relocated.

When *init* comes up initially and whenever it switches out of *SINGLE USER* state to normal run states, it sets the *ioctl*(2) states of the virtual console, **/dev/syscon**, to those modes saved in the file **/etc/ioctl.syscon**. This file is written by *init* whenever *SINGLE USER* mode is entered. If this file doesn't exist when *init* wants to read it, a warning is printed and default settings are assumed.

If a **0** through **6** is entered *init* enters the corresponding *run-level*. Any other input will be rejected and the user will be re-prompted. If this is the first time *init* has entered a *run-level* other than *SINGLE USER*, *init* first scans *inittab* for special entries of the type *boot* and *bootwait*. These entries are performed, providing the *run-level* entered matches that of the entry before any normal processing of *inittab* takes place. In this way any special initialization of the operating system,such as mounting file systems, can take place before users are allowed onto the system. The *inittab* file is scanned to find all entries that are to be processed for that *run-level*.

Run-level **2** is usually defined by the user to contain all of the terminal processes and daemons that are spawned in the multi-user environment.

In a multi-user environment, the *inittab* file is usually set up so that *init* will create a process for each terminal on the system.

For terminal processes, ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as the result of hanging up. When *init* receives a child death signal, telling it that a process it spawned has died, it records the fact and the reason it died in **/etc/utmp** and **/etc/wtmp** if it exists (see *who*(1)). A history of the processes spawned is kept in **/etc/wtmp** if such a file exists.

To spawn each process in the *inittab* file, *init* reads each entry and for each entry which should be respawned, it forks a child process. After it has spawned all of the processes specified by the *inittab* file, *init* waits for one of its descendant processes to die, a powerfail signal, or until *init* is signaled by *init* or *telinit* to change the system's *run-level.* When one of the above three conditions occurs, *init* re-examines the *inittab* file. New entries can be added to the *inittab* file at any time; however, *init* still waits for one of the above three conditions to occur. To provide for an instantaneous response the **init Q** or **init q** command can wake *init* to re-examine the *inittab* file.

If *init* receives a *powerfail* signal (*SIGPWR*) and is not in *SINGLE USER* mode, it scans *inittab* for special powerfail entries. These entries are invoked (if the *run-levels* permit) before any further processing takes place. In this way *init* can perform various cleanup and recording functions whenever the operating system experiences a power failure.

When *init* is requested to change *run-levels* (via *telinit*), *init* sends the warning signal (**SIGTERM**) to all processes that are undefined in the target *run-level*. *Init* waits 20 seconds before forcibly terminating these processes via the kill signal (**SIGKILL**).

Telinit

Telinit, which is linked to */etc/init*, is used to direct the actions of *init*. It takes a one character argument and signals *init* via the kill system call to perform the appropriate action. The following arguments serve as directives to *init*.

0–6 tells *init* to place the system in one of the *run-levels* **0–6**.

a,**b**,**c** tells *init* to process only those **/etc/inittab** file entries having the **a**, **b** or **c** *run-level* set.

Q,**q** tells *init* to re-examine the **/etc/inittab** file.

s,**S** tells *init* to enter the single user environment. When this level change is effected, the virtual system teletype, **/dev/syscon**, is changed to the terminal from which the command was executed.

Telinit can only be run by someone who is super-user or a member of group **sys**.

FILES

/etc/inittab
/etc/utmp
/etc/wtmp
/etc/ioctl.syscon
/dev/syscon
/dev/systty

SEE ALSO

getty(1M), login(1M), sh(1), who(1), kill(2), inittab(4), utmp(4).

DIAGNOSTICS

If *init* finds that it is continuously respawning an entry from **/etc/inittab** more than 10 times in 2 minutes, it will assume that there is an error in the command string, and generate an error message on the system console, and refuse to respawn this entry until either 5 minutes has elapsed or it receives a signal from a user *init* (*telinit*). This prevents *init* from eating up system resources when someone makes a typographical error in the *inittab* file or a program is removed that is referenced in the *inittab*.

NAME

iv – initialize and maintain volume

SYNOPSIS

iv [**-iustdwmv**] *special* [*descriptionfile*]

DESCRIPTION

Iv initializes and maintains a UNIX PC disk volume. *Special* and *descriptionfile* specify the disk and a description file for it; these are described below. *Iv* does one of five operations, specified by the following options:

−i completely initialize a volume. This consists of five phases:

1. Initialize *iv*'s internal Volume Home Block, based on *descriptionfile* and the disk type. If the disk can support bad block handling (all types except floppies), create an internal Bad Block Table. Put bad block data from *descriptionfile* and volume's existing Bad Block Table (if any) in internal Bad Block Table.
2. Format medium.
3. Perform a surface check. If the disk can support bad block handling, add bad blocks to the Bad Block Table. If the disk cannot support bad block handling, the first bad spot causes the disk to be rejected.
4. Write out the Volume Home Block. This has the effect of dividing the volume into slices (partitions).
5. Allocate and write out the files that share the Reserved Area (slice 0) with the Volume Home Block. If the disk can support bad block handling, one of these files is the Bad Block Table. Other files are specified in *descriptionfile*.

−u Update the volume home block. This is the same as −i except that the second and third phases (medium formatting and surface check) are skipped.

−s Surface test. Any bad blocks discovered are added to the bad block table.

−t Tell volume description. Display volume home block in human-readable form. No description file is needed. The volume's contents are not affected.

−d Description file display. A description file that describes the current state of the volume is written to the standard output. If the Reserved Area contains a loader, the loader keyword's value is written as **/usr/lib/iv/loader**. If the Reserved Area contains a down load image area, the Down Load Area Description lists files whose names are of the form

/usr/lib/iv/wsxxx.yyy

Where *xxx* is the numeric device identification; and *yyy* is **422** if *xxx* is even, **232** if *xxx* is odd.

The **−f** option, equivalent to **−u**, is provided for compatibility with older versions of *iv*. It should not be used, as it may disappear in future releases.

In addition to the single operation option (**−i, −u, −s, −t,** or **−d**), you can specify any or all of the following options:

−v Verbose display output. If the display includes The Volume Home Block, also include the bad block table.

−l A normal surface test consists of a single pass over the disk; **−l** specifies ten passes.

−w A normal surface test pass consists of a read pass; **− w** specifies a write pass before each read pass.

File Parameters

Special is the character special file for slice zero on the volume. This name takes the form **/dev/rfp0*t*0**, where *t* is 0 for winchester, 2 for floppy.

Descriptionfile is a text file that describes the volume. It is required by the **−i** and **−u** options. The description file consists of four parts:

general disk description
reserved area files description
bad blocks description
partition table description

The four descriptions are separated from each other by four lines, each of which contains only a single dollar sign ($). Specifics for each of the five descriptions are given under separate headings below.

General Description

Each line in the General Description begins with a keyword. Some keywords are followed by values; the value is separated from the keyword by spaces or tabs. For example:

hitech
cylinders 25

Each keyword is only used once. Here are the valid keywords.

type Mandatory, unless the volume is already initialized in UNIX PC format. Value is disktype: **HD** for winchester, and **FD** for floppy.

name Mandatory, unless the volume is already initialized in UNIX PC format. Value is the volume name. Any characters except spaces or tabs are permitted in the volume name. The actual name in the Volume Home Block is always exactly six characters; *iv* right truncates names that are too long and right pads with nulls names that are too short.

cylinders
Mandatory, unless the volume is already initialized in UNIX PC format. Value is the number of cylinders on the

disk. This must be a positive number not greater than 1024.

heads Mandatory, unless the volume is already initialized in UNIX PC format. Value is the number of heads on the disk. This must be a positive number not greater than 7.

sectors
Mandatory, unless the volume is already initialized in UNIX PC format. Value is the number of physical sectors per track.

steprate
Mandatory, unless the volume is already initialized in UNIX PC format. Value is a number that is passed to the disk controller. Currently this number must be 0.

exchangeable
If this keyword is present, the disk can be removed from its drive (floppy).

hitech If this keyword is present, write precompensation is not required on the disk. See the disk manufacturer's documentation for further information.

Reserved Area Description

The Reserved Area Description describes the files that share slice zero with the volume home block. Each line in the Reserved Area Description consists of a keyword followed by one or more parameters; one or more tabs or spaces separates keywords and parameters from each other. Here are the valid keywords and their meanings. (A logical block is 1024 bytes long.)

loader Describes the loader area. The first, mandatory, parameter is the full pathname of an **a.out** file to put in the loader area. The second, optional, parameter is the size of the loader area in logical blocks. If the second parameter is missing, the size of the **a.out** file is used.

badblocktable
Describes the bad block table. The first, mandatory, parameter is the size of the bad block table in logical blocks. The second, optional, parameter is only used when an existing bad block table contains errors; this parameter is "empty" to clear the bad block table, missing otherwise.

All lines valid for the Reserved Area Files Description are optional. However, the bad block table is mandatory on a volume which supports bad block handling, and the loader area is mandatory on a volume which is to hold an operating system. A system volume cannot have a bootable program area.

Bad Blocks Description

The Bad Block Description explicitly specifies up to 255 bad blocks to be added to the bad blocks table. *Iv* merges specified bad block information with information already in the bad block table (if there already is one) and bad block information discovered through the surface test.

Each bad block entry is a single line. There are two forms: *s* where *s* is a sector number; *c h b* where *c* is a cylinder number, *h* is a head number, and *b* is a byte number. Both forms condemn a single sector, the second the sector that contains the specified byte.

The last sector on each track serves as a bad block alternate. *Iv* chooses its alternates in a way that minimizes extra seeking for alternate blocks.

Partition Table Description

The Partition Table Description shows where the slices (partitions) are to begin. Each line in the file consists of a track number, the starting track of a slice. Slices are listed in ascending numeric order and begin on successively higher tracks. The beginning of a slice defines the end of the previous slice. The first track number is always 0, since slice zero always begins on a track zero.

There can be at most 16 slices on a disk. It is a fatal error to specify a slice one that doesn't leave enough room in slice zero for the Volume Home Block and the slice zero files.

EXAMPLES

Here is an example of a disk description file.

```
#      iv description file for standard floppy disk
type FD
name Data
cylinders 40
heads    2
sectors  8
steprate0
$
$
$
0
1
```

FILES

/dev/rfp*   disk character special files
/usr/lib/iv/*   prototype description files

SEE ALSO

dismount(1M), update(1), gd(7).

WARNINGS

The **-i**, **-u**, and **-s** operations are dangerous or fatal to existing volume data. Always precede these operations with a backup.

When a new bad block is itself an alternate block, *iv* may produce messages that appear spurious but are actually correct. If the bad block is already in use as an alternate, the "added bad block" message can appear twice for one block.

NAME

killall – kill all active processes

SYNOPSIS

/etc/killall [signal]

DESCRIPTION

Killall is is a procedure used by **/etc/shutdown** to kill all active processes not directly related to the shut down procedure.

Killall is chiefly used to terminate all processes with open files so that the mounted file systems will be unbusied and can be unmounted.

Killall sends *signal* (see *kill*(1)) to all remaining processes not belonging to the above group of exclusions. If no *signal* is specified, a default of **9** is used.

FILES

/etc/shutdown

SEE ALSO

kill(1), ps(1), shutdown(1M), signal(2).

NAME

lddrv – manage loadable drivers

SYNOPSIS

lddrv [**–m** master] [**–abdqsuv**] [devname]
lddrv –a [v] [**–m** master] [**–o** dfile] devname
lddrv –d [v] [**–m** master] devname
lddrv –b [v] [**–m** master] devname
lddrv –u [v] [**–m** master] devname
lddrv –q [v] [**–m** master] devname
lddrv –s [v] [**–m** master]

DESCRIPTION

Lddrv allocates/deallocates space for a specified driver, loads/unloads a specified driver, and returns the status of specified driver(s).

The *v* argument writes driver information to stdout. Without the *v* argument, *lddrv* is silent, even when an error occurs. **–m** *master* specifies the name of the master file to be used for this particular *lddrv* run (default is /etc/master). Use **–o** *dfile* to specify the name of the file that contains the driver's executable code, if the name of this file is different from the driver name. The *devname* argument is the name of the driver.

The options are:

–a Allocate space for and load the driver.

–d Unload the driver and deallocate its space.

–b Load (bind) the driver.

–u Unload (unbind) the driver.

–q Return the status of a particular loadable driver.

–s Return the status of all loadable drivers.

The first time **lddrv -a** is run for a new or updated ".o" executable file, the unresolved kernel symbol references are resolved using the ascii kernel symbol table file **/etc/lddrv/unix.sym** A file containing the executable with all symbols resolved is written to a file whose name is the driver name.

EXAMPLES

To show the following status report, use this format of *lddrv* : **lddrv -s**

DEVNAME	ID	BLK	CHAR	LINE	SIZE	ADDR	FLAGS
lipc	0	–1	1	–1	0x5000	0x3dd000	ALLOC BOUND
plp	1	–1	6	–1	0x1000	0x3e2000	ALLOC BOUND
xt	2	–1	9	1	0x3000	0x3e3000	ALLOC BOUND

To allocate and load the FPA driver and write binding information to stdout, use this format of *lddrv* : **lddrv -av fpa**

To silently unload the FPA driver but leave its memory allocated, use this format of *lddrv* : **lddrv -u fpa**

To load (bind) the FPA driver if it is already allocated, use this format of *lddrv* : **lddrv -bv fpa**

To unload the FPA driver and deallocate its memory space, use this format of *lddrv* : **lddrv -db fpa**

To allocate and load the driver **foo** whose executable code is in **foobar.o**, use this format of *lddrv* : **lddrv -av -o foobar.o foo**

FILES

/etc/master	default master file
/etc/drvtab	loadable driver table
/etc/lddrv	directory that contains *lddrv* and loadable drivers
/etc/lddrv/drivers	a list of drivers to be loaded automatically during reboot, one driver name per line.

SEE ALSO

syslocal(2), master(4), drivers(7).

NAME

login – sign on

SYNOPSIS

login [name [env-var . . .]]

DESCRIPTION

The *login* command is used at the beginning of each terminal session and allows you to identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a *cntrl-d* to indicate an "end-of-file." (See *How to Get Started* at the beginning of this volume for instructions on how to dial up initially.)

If *login* is invoked as a command it must replace the initial command interpreter. This is accomplished by typing:

    exec login

from the initial shell.

Login asks for your user name (if not supplied as an argument), and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it will not appear on the written record of the session.

At some installations, an option may be invoked that will require you to enter a second "dialup" password. This will occur only for dial-up connections, and will be prompted by the message "dialup password:". Both passwords are required for a successful login.

If you do not complete the login successfully within a certain period of time (e.g., one minute), you are likely to be silently disconnected.

After a successful login, accounting files are updated, the procedure */etc/profile* is performed, the message-of-the-day, if any, is printed, the user-ID, the group-ID, the working directory, and the command interpreter (usually *sh*(1)) are initialized, and the file **.profile** in the working directory is executed, if it exists. These specifications are found in the **/etc/passwd** file entry for the user. The name of the command interpreter is – followed by the last component of the interpreter's pathname (i.e., **–sh**). If this field in the password file is empty, then the default command interpreter, */bin/sh* is used.

The basic *environment* (see *environ*(5)) is initialized to:

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
TZ=*timezone-specification*

The environment may be expanded or modified by supplying additional arguments to *login*, either at execution time or when *login* requests your login name. The arguments may take either the form *xxx* or *xxx=yyy*. Arguments without an equal sign are placed in the environment as

L*n*=xxx

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variables containing an = are placed into the environment without modification. If they already appear in the environment, then they replace the older value. There are two exceptions. The variables **PATH** and **SHELL** cannot be changed. This prevents people, logging into restricted shell environments, from spawning secondary shells which aren't restricted. Both *login* and *getty* understand simple single character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

FILES

/etc/utmp	accounting
/etc/wtmp	accounting
/usr/mail/*your-name*	mailbox for user *your-name*
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/profile	system profile
.profile	user's login profile

SEE ALSO

mail(1), newgrp(1), sh(1), su(1), passwd(4), profile(4), environ(5).

DIAGNOSTICS

Login incorrect if the user name or the password cannot be matched.

No shell, cannot open password file, or *no directory*: consult a UNIX programming counselor.

No utmp entry. You must exec "login" from the lowest level "sh". if you attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

NAME

lpadmin – configure the LP spooling system

SYNOPSIS

/usr/lib/lpadmin **−p** printer [options]
/usr/lib/lpadmin **−x** dest
/usr/lib/lpadmin **−d**[dest]

DESCRIPTION

Lpadmin configures LP spooling systems to describe printers, classes and devices. It is used to add and remove destinations, change membership in classes, change devices for printers, change printer interface programs and to change the system default destination. *Lpadmin* may not be used when the LP scheduler, *lpsched*(1M), is running, except where noted below.

Exactly one of the **−p**, **−d** or **−x** options must be present for every legal invocation of *lpadmin*.

−d[*dest*] makes *dest*, an existing destination, the new system default destination. If *dest* is not supplied, then there is no system default destination. This option may be used when *lpsched*(1M) is running. No other *options* are allowed with **−d**.

−x*dest* removes destination *dest* from the LP system. If *dest* is a printer and is the only member of a class, then the class will be deleted, too. No other *options* are allowed with **−x**.

−p*printer* names a *printer* to which all of the *options* below refer. If *printer* does not exist then it will be created.

The following *options* are only useful with **−p** and may appear in any order. For ease of discussion, the printer will be referred to as *P* below.

−c*class* inserts printer *P* into the specified *class*. *Class* will be created if it does not already exist.

−e*printer* copies an existing *printer's* interface program to be the new interface program for *P*.

−h indicates that the device associated with *P* is hardwired. This *option* is assumed when creating a new printer unless the **−l** *option* is supplied.

−i*interface* establishes a new interface program for *P*. *Interface* is the path name of the new program.

−l indicates that the device associated with *P* is a login terminal. The LP scheduler, *lpsched*, disables all login terminals automatically each time it is started. Before re-enabling *P*, its current *device* should be established using *lpadmin*.

−m*model* selects a model interface program for *P*. *Model* is one of the model interface names supplied with the LP software (see *Models* below).

−r*class* removes printer *P* from the specified *class*. If *P* is the last member of the *class*, then the *class* will be removed.

−v*device* associates a new *device* with printer *P*. *Device* is the pathname of a file that is writable by the LP administrator, *lp*. Note that there is nothing to stop an administrator from associating the same *device* with more than one *printer*. If only the **−p** and **−v** *options* are supplied, then *lpadmin* may be used while the scheduler is running.

Restrictions.

When creating a new printer, the **−v** option and one of the **−e**, **−i** or **−m** options must be supplied. Only one of the **−e**, **−i** or **−m** options may be supplied. The **−h** and **−l** keyletters are mutually exclusive. Printer and class names may be no longer than 14 characters and must consist entirely of the characters **A-Z**, **a-z**, **0-9** and _ (underscore).

Models.

Model printer interface programs are supplied with the LP software. They are shell procedures which interface between *lpsched* and devices. All models reside in the directory **/usr/spool/lp/model** and may be used as is with *lpadmin* **−m**. Alternatively, LP administrators may modify copies of models and then use *lpadmin* **−i** to associate them with printers. The following list describes the *models* and lists the options which they may be given on the *lp* command line using the **−o** keyletter.

imagen_S For Imagen serial page printer

dumb For dumb parallel line printer

dumb_remote For printer in remote mode. Option:

RAW The request is printed in raw mode (no post-processing, no CR-LF translation, high-order bits are passed through unchanged.)

second_remote Modification of **dumb_remote** to support the second remote printer. Option:

RAW The request is printed in raw mode (no post-processing, no CR-LF translation, high-order bits are passed through unchanged.)

dumb_S For dumb serial line printer. Option:

RAW The request is printed in raw mode (no post-processing, no CR-LF translation, high-order bits are passed through unchanged.)

n450 Modification of **dumb_S** so that NL will not map to CR-NL. This model is useful when the user wants to use **neqn** coupled with **nroff** on a Diablo 450 or compatible printer. Option:

RAW The request is printed in raw mode (no post-processing, no CR-LF translation, high-order bits are passed through unchanged.)

EXAMPLES

1. Assuming there is an existing AT&T 475 line printer named *ATT475*, it will use the **dumb_S** model interface after the command:

   /usr/lib/lpadmin -pATT475 -mdumb_S

2. To obtain raw mode printing on *ATT475*, use the command:

   lp -dATT475 -oRAW files

3. A Diablo 450 printer called *st2* can be added to the LP configuration using the command:

   /usr/lib/lpadmin -pst2 -v/dev/tty001 -mn450

4. A Diablo 1640 printer called *st1* can be added to the LP configuration (Note: interface model 1640 is currently not available on the UNIX PC). Use the command:

   /usr/lib/lpadmin -pst1 -v/dev/tty002 -mdumb_s

5. An *nroff* document may be printed on *st1* in any of the following ways:

   nroff -T450 files | lp -dst1 -of
   nroff -T450-12 files | lp -dst1 -of
   nroff -T37 files | col | lp -dst1

6. The following command prints the password file on *st1* in 12-pitch:

   lp -dst1 -o12 /etc/passwd

   NOTE: the **-12** option to the **1640** model should never be used in conjunction with *nroff*.

FILES

/usr/spool/lp/*

SEE ALSO

450(1), accept(1M), enable(1), lp(1), lpsched(1M), lpstat(1).

BUGS

The -o option is not available in Version 3.5.

NAME

lpsched, lpshut, lpmove – start/stop the LP request scheduler and move requests

SYNOPSIS

/usr/lib/lpsched
/usr/lib/lpshut
/usr/lib/lpmove requests dest
/usr/lib/lpmove dest1 dest2

DESCRIPTION

Lpsched schedules requests taken by *lp*(1) for printing on line printers.

Lpshut shuts down the line printer scheduler. All printers that are printing at the time *lpshut* is invoked will stop printing. Requests that were printing at the time a printer was shut down will be reprinted in their entirety after *lpsched* is started again. All LP commands perform their functions even when *lpsched* is not running.

Lpmove moves requests that were queued by *lp*(1) between LP destinations. This command may be used only when *lpsched* is not running.

The first form of the command moves the named *requests* to the LP destination, *dest*. *Requests* are request ids as returned by *lp*. The second form moves all requests for destination *dest1* to destination *dest2*. As a side effect, *lp* will reject requests for *dest1*.

Note that *lpmove* never checks the acceptance status (see *accept*(1M)) for the new destination when moving requests.

FILES

/usr/spool/lp/*

SEE ALSO

accept(1M), enable(1), lp(1), lpadmin(1M), lpstat(1).

NAME

masterupd – update the /etc/master file

SYNOPSIS

masterupd [**–abcdl**] [**–m** *master*] *flags devname*

DESCRIPTION

Masterupd is used to manage the **/etc/master** file. Using *masterupd* you can add entries, delete entries, list entries, or find the block or character major device numbers for device *devname*.

The options are as follows:

–a Add an entry to the master table.

–b Get block device number for device *devname*. Returns the number as a string on standard output.

–c Get character device number for device *devname*. Returns the number as a string on standard output.

–d Delete the entry for device *devname* from the master table.

–l List the entries in the master table.

Use **–m** *master* to specify a different *master* file than the default **/etc/master**. The *flags* are used only with the **–a** (add) option, and may be any of:

–p *prefix*	subroutine prefix name (if different from *devname*)
–B *number*	force a particular block major number
–C *number*	force a particular character major number
block	device is a block device
char	device is a character device
init	device has an init routine
release	device has a release routine
open	device has an open routine
close	device has a close routine
read	device has a read routine
write	device has a write routine
ioctl	device has an ioctl routine
strategy	device has a strategy routine
print	device has a print routine
pwr	device has a power failure handler routine
lopen	device has a line disipline open routine
lclose	device has a line disipline close routine
lread	device has a line disipline read routine
lwrite	device has a line disipline write routine
lioctl	device has a line disipline ioctl routine
linput	device has a line disipline input routine
loutput	device has a line disipline output routine
lmdmint	device has a line disipline modem interrupt routine
fix	device has a fixed vector
flt	device has a floating vector

required	device is a required device
supp	suppress interrupt vector
nocnt	suppress count field in conf.c
once	allow only one of these devices
info	device has an info routine

EXAMPLES

To add an entry for a loadable parallel printer:

```
masterupd -a -C 6 -p lp open close
        init release write ioctl
        char plp
```

To add an entry for a loadable ipc:

```
masterupd -a init release lipc
```

To delete the above entry:

```
masterupd -d lipc
```

Below is an Install script fragment to install a new character device 'xyzzy':

```
        .
        .
        .
masterupd -a -p xy_ init release open close read write ioctl
char xyzzy
charnum=`masterupd -c xyzzy`
for i in 1 2 3 4 5 6
do
        /etc/mknod /dev/xy$i c $charnum $i
done
```

The above example results in the following entry:

```
xyzzy   1137   000   004   xy_   0   11
```

The following is an Install script fragment that includes some loadable line discipline flags to add an entry to the master table.

```
        .
        .
        .
masterupd -a  char release open close read write ioctl
linput loutput xt
        .
        .
        .
```

The above example results in the following entry:

```
xt   1037   140   004   xt   0   10
```

SEE ALSO

drivers(7), lddrv(1M), master(4).

NAME

mkfs – construct a file system

SYNOPSIS

/etc/mkfs special blocks[:inodes] [gap blocks/cyl]
/etc/mkfs special proto [gap blocks/cyl]
/etc/mkfs special

DESCRIPTION

Mkfs constructs a file system by writing on the special file according to the directions found in the remainder of the command line. If the second argument is given as a string of digits, *mkfs* builds a file system with a single empty directory on it. The size of the file system is the value of *blocks* interpreted as a decimal number. This is the number of *physical* disk blocks the file system will occupy. The boot program is left uninitialized. If the optional number of inodes is not given, the default is the number of *logical* blocks divided by 4.

If the second argument is a file name that can be opened, *mkfs* assumes it to be a prototype file *proto*, and will take its directions from that file. The prototype file contains tokens separated by spaces or new-lines. The first token is the name of a file to be copied onto block zero as the bootstrap program. The second token is a number specifying the size of the created file system in *physical* disk blocks. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of inodes in the file system. The maximum number of inodes configurable is 65500. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user ID, the group ID, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **-bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or – to specify set-user-id mode or not. The third is **g** or – for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod*(1)).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a path name whence the contents and size are copied. If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers. If the file is a directory, *mkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token $.

A sample prototype specification follows:

```
/stand/diskboot
4872 110
d--777 3 1
usr     d--777 3 1
        sh      ---755 3 1 /bin/sh
        ken     d--755 6 1
                $
        b0      b--644 3 1 0 0
        c0      c--644 3 1 0 0
        $
$
```

The third form of the command syntax is recommended, since it needs no parameters, just special file.

The *default* will be used if the supplied *gap* and *blocks/cyl* are considered illegal values or if a short argument count occurs.

SEE ALSO

dir(4), fs(4).

BUGS

If a prototype is used, it is not possible to initialize a file larger than 64K bytes, nor is there a way to specify links.

NAME

mknod – build special file

SYNOPSIS

/etc/mknod name **c | b** major minor
/etc/mknod name **p**

DESCRIPTION

Mknod makes a directory entry and corresponding i-node for a special file. The first argument is the *name* of the entry. In the first case, the second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number), which may be either decimal or octal.

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file **conf.c**.

Mknod can also be used to create fifo's (a.k.a named pipes) (second case in *SYNOPSIS* above).

SEE ALSO

mknod(2).

NAME

mount, umount – mount and dismount file system

SYNOPSIS

/etc/mount [special directory [**−r**]]

/etc/umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The *directory* must exist already; it becomes the name of the root of the newly mounted file system.

These commands maintain a table of mounted devices. If invoked with no arguments, *mount* prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

FILES

/etc/mnttab   mount table

SEE ALSO

setmnt(1M), mount(2), mnttab(4).

DIAGNOSTICS

Mount issues a warning if the file system to be mounted is currently mounted under another name.

Umount complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or some user's working directory.

BUGS

Some degree of validation is done on the file system, however it is generally unwise to mount garbage file systems.

NAME

ncheck – generate names from i-numbers

SYNOPSIS

/etc/ncheck [**–i** numbers] [**–a**] [**–s**] [file-system]

DESCRIPTION

Ncheck with no argument generates a path name vs. i-number list of all files on a set of default file systems. Names of directory files are followed by **/.**. The **–i** option reduces the report to only those files whose i-numbers follow. The **–a** option allows printing of the names **.** and **..**, which are ordinarily suppressed. The **–s** option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

The report is in no useful order, and probably should be sorted.

SEE ALSO

fsck(1M), sort(1).

DIAGNOSTICS

When the file system structure is improper, **??** denotes the "parent" of a parentless file and a path name beginning with **...** denotes a loop.

NAME

rc – system initialization shell script

SYNOPSIS

/etc/rc

DESCRIPTION

This shell procedure is executed via */etc/inittab* by *init*(1M) when the system state is changed.

The *rc* procedure clears the mounted file system table, */etc/mnttab* (see *mnttab*(4)), performs all the necessary consistency checks to prepare the system to change into multi-user mode.

The *rc* procedure starts all system daemons before the terminal lines are enabled. In addition, file systems are mounted and accounting, window, status, and telephony management is started.

SEE ALSO

init(1M), shutdown(1M), inittab(4).

NAME

reboot – reboot the system

SYNOPSIS

/etc/reboot

DESCRIPTION

Reboot causes the processor to enter its system bootstrap code thereby rebooting the system.

Reboot will enter the boot sequence immediately, without flushing the internal system buffers. It must be used with extreme caution.

SEE ALSO

sync(1M).

NAME

setmnt – establish mount table

SYNYOPSIS

/etc/setmnt

DESCRIPTION

Setmnt creates the **/etc/mnttab** table (see *mnttab*(4)), which is needed for both the *mount*(1M) and *umount* commands. *Setmnt* reads the standard input and creates a mnttab entry for each line. Input lines have the format:

filesys node

where *filesys* is the name of the file system's special file (e.g., **rfp***) and *node* is the root name of that file system. Thus *filesys* and *node* become the first two strings in the *mnttab*(4) entry.

FILES

/etc/mnttab

SEE ALSO

mnttab(4).

BUGS

Evil things will happen if *filesys* or *node* is longer than 10 characters.

Setmnt silently enforces an upper limit on the maximum number of mnttab entries.

NAME

sfont, setf – install or load font

SYNOPSIS

sfont [–**s**] [fontfile] slotnumber
setf [–**s**] [fontfile] slotnumber

DESCRIPTION

There are 16 system slots and 8 window slots available. A font must be loaded into a window slot to be selectable. Each font loaded into a window slot is also *installed* in a system slot to avoid having multiple copies of that font.

Sfont allows a font to be installed (or deinstalled) into the system slot (number 0 through 15). If the *slot number* is 0 through 7, then windows created subsequently will inherit the font in the corresponding window slot. If the *slot number* is 8 through 15, the font will be installed in the system but not known until a window loads it.

Setf loads (or unloads) a font into the window slot (number 0 through 7) of the current window immediately and exclusively.

The *–s* option specifies silent mode. Any errors are reported by return value.

Omitting the *font file* argument with either command causes the *slot number* to be unloaded or deinstalled.

FILES

/usr/lib/wfont
/etc/sfont
/etc/setf

SEE ALSO

syslocal(2), font(4), escape(7), window(7)

NAME

shutdown – terminate all processing

SYNOPSIS

/etc/shutdown

DESCRIPTION

Shutdown is a shell script that accomplishes the following:

1. Kills all processes (*killall*).
2. Stops the *lp* scheduler (*lpshut*).
3. Transfers control to */etc/profile* (*init*), which recognizes that it is in single-user mode and reboots.

SEE ALSO

mount(1M), sync(1).

NAME

uucico – file transport program for the uucp system

SYNOPSIS

/usr/lib/uucp/uucico [**–r** role_number] [**–x** debug_level] **–s** system_name

DESCRIPTION

Uucico is the file transport program for *uucp* work file transfers. Role numbers for the **–r** option are the digit 1 for master mode or 0 for slave mode (default). The **–r** option should be specified as the digit 1 for master mode when *uucico* is started by a user, program, or *cron*. *Uux* and *uucp* both queue jobs that will be transferred by *uucico*.

The **–x** option sets the level of debugging output. If **–s** is specified, then a call to *system_name* is made even if there is no work for that system in the spool directory. Calls are only made during the times permitted in **/usr/lib/uucp/L.sys**. This can be used to poll sites that cannot initiate a connection.

FILES

/usr/lib/uucp/modemcap
/usr/lib/uucp/L.sys
/usr/lib/uucp/L-devices
/usr/lib/uucp/L-dialcodes
/usr/spool/uucp*
/usr/spool/uucpublic/*

SEE ALSO

cron(1M), uucp(1C), uustat(1C), uux(1C), uuclean(1M), uusub(1M).

NAME

uuclean – uucp spool directory clean-up

SYNOPSIS

/usr/lib/uucp/uuclean [options]

DESCRIPTION

Uuclean will scan the spool directory for files with the specified prefix and delete all those which are older than the specified number of hours.

The following options are available.

−d*directory*	Clean *directory* instead of the spool directory.
−p*pre*	Scan for files with *pre* as the file prefix. Up to 10 **−p** arguments may be specified. A **−p** without any *pre* following will cause all files older than the specified time to be deleted.
−n*time*	Files whose age is more than *time* hours will be deleted if the prefix test is satisfied. (default time is 72 hours)
−w*file*	The default action for *uuclean* is to remove files which are older than a specified time (see **−n** option). The **−w** option is used to find those files older than *time* hours, however, the files are not deleted. If the argument *file* is present the warning is placed in *file*, otherwise, the warnings will go to the standard output.
−s*sys*	Only files destined for system *sys* are examined. Up to 10 **−s** arguments may be specified.
−m*file*	The **−m** option sends mail to the owner of the file when it is deleted. If a *file* is specified then an entry is placed in *file*.

This program is typically started by *cron*(1M).

FILES

/usr/lib/uucp	directory with commands used by *uuclean* internally
/usr/spool/uucp	spool directory

SEE ALSO

cron(1M), uucp(1C), uux(1C).

NAME

uusub – monitor uucp network

SYNOPSIS

/usr/lib/uucp/uusub [options]

DESCRIPTION

Uusub defines a *uucp* subnetwork and monitors the connection and traffic among the members of the subnetwork. The following options are available:

−**a***sys*	Add *sys* to the subnetwork.
−**d***sys*	Delete *sys* from the subnetwork.
−**l**	Report the statistics on connections.
−**r**	Report the statistics on traffic amount.
−**f**	Flush the connection statistics.
−**u***hr*	Gather the traffic statistics over the past *hr* hours.
−**c***sys*	Exercise the connection to the system *sys*. If *sys* is specified as **all**, then exercise the connection to all the systems in the subnetwork.

The meanings of the connections report are:

sys #call #ok time #dev #login #nack #other

where *sys* is the remote system name, *#call* is the number of times the local system tries to call *sys* since the last flush was done, *#ok* is the number of successful connections, *time* is the latest successful connect time, *#dev* is the number of unsuccessful connections because of no available device (e.g. ACU), *#login* is the number of unsuccessful connections because of login failure, *#nack* is the number of unsuccessful connections because of no response (e.g. line busy, system down), and *#other* is the number of unsuccessful connections because of other reasons.

The meanings of the traffic statistics are:

sfile sbyte rfile rbyte

where *sfile* is the number of files sent and *sbyte* is the number of bytes sent over the period of time indicated in the latest *uusub* command with the −**u***hr* option. Similarly, *rfile* and *rbyte* are the numbers of files and bytes received.

The command:

uusub −c all −u 24

is typically started by *cron*(1M) once a day.

FILES

/usr/spool/uucp/SYSLOG	system log file
/usr/lib/uucp/L_sub	connection statistics
/usr/lib/uucp/R_sub	traffic statistics

SEE ALSO

uucp(1C), uustat(1C).

NAME

volcopy, labelit – copy file systems with label checking

SYNOPSIS

/etc/volcopy [options] fsname special1 volname1 special2 volname2

/etc/labelit special [fsname volume [**–n**]]

DESCRIPTION

Volcopy makes a literal copy of the file system using a blocksize matched to the device. *Options* are:

–a	invoke a verification sequence requiring a positive operator response instead of the standard 10 second delay before the copy is made,
–s	(default) invoke the **DEL if wrong** verification sequence.

Other *options* are used only with tapes:

–**bpi**density	bits-per-inch (i.e., **800/1600/6250**),
–**feet**size	size of reel in feet (i.e., **1200/2400**),
–**reel**num	beginning reel number for a restarted copy,
–buf	use double buffered I/O.

The program requests length and density information if it is not given on the command line or is not recorded on an input tape label. If the file system is too large to fit on one reel, *volcopy* will prompt for additional reels. Labels of all reels are checked. Tapes may be mounted alternately on two or more drives.

The *fsname* argument represents the mounted name (e.g.: **root**, **u1**, etc.) of the filsystem being copied.

The *special* should be the physical disk section or tape (e.g.: **/dev/rdsk15**, **/dev/rmt0**, etc.).

The *volname* is the physical volume name (e.g.: **pk3**, **t0122**, etc.) and should match the external label sticker. Such label names are limited to six or fewer characters. *Volname* may be – to use the existing volume name.

Special1 and *volname1* are the device and volume from which the copy of the file system is being extracted. *Special2* and *volname2* are the target device and volume.

Fsname and *volname* are recorded in the last 12 characters of the superblock (**char fsname[6], volname[6];**).

Labelit can be used to provide initial labels for unmounted disk or tape file systems. With the optional arguments omitted, *labelit* prints current label values. The **–n** option provides for initial labeling of new tapes only (this destroys previous contents).

FILES

/etc/log/filesave.log   a record of file systems/volumes copied

SEE ALSO

fs(4).

BUGS

Only device names beginning **/dev/rmt** (on DEC systems) or **/dev/rtp** (on 3B20S systems) are treated as tapes.

NAME

wall – write to all users

SYNOPSIS

/etc/wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by:

Broadcast Message from **. . .**

It is used to warn all users, typically prior to shutting down the system.

The sender must be super-user to override any protections the users may have invoked (see *mesg*(1)).

FILES

/dev/tty*

SEE ALSO

mesg(1), write(1).

DIAGNOSTICS

"*Cannot send to* . . ." when the open on a user's tty file fails.

NAME

whodo – who is doing what

SYNOPSIS

/etc/whodo

DESCRIPTION

Whodo produces merged, reformatted, and dated output from the *who*(1) and *ps*(1) commands.

SEE ALSO

ps(1), who(1).

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals and UNIX device drivers. The names of the entries are generally derived from names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding UNIX device driver are discussed where applicable.

BUGS

While the names of the entries *generally* refer to vendor hardware names, in certain cases these names are seemingly arbitrary for various historical reasons.

NAME

drivers – loadable device drivers

DESCRIPTION

The following information should be taken into consideration when writing loadable drivers.

A loadable driver is like a fixed, linked-in device driver. It has access to all kernel subroutines and global data. After it is loaded, it is effectively part of the running kernel.

Differences between loadable and ordinary drivers involve their init routines, release routines, and interrupt processing.

Init Routines

Loadable drivers may have an init routine that is executed when the driver is bound, and a release routine that is executed when the driver is unbound (see *lddrv*(1M) for a description of driver allocation and bind operations. Init routines check for the existence of hardware, initialize the hardware, put the interrupt service routine for the hardware into the interrupt chain, and do other similar tasks.

Release Routines

Release routines make sure the device or driver is idle, turn off the device, take the interrupt service routine out of the interrupt chain, and similar tasks. A typical action for a release routine to take when the device *is not* idle is to set an error code in **u_u.error** and return. If the device is *guaranteed* to become idle in a limited amount of time, the routine may do a *sleep()* (see *lprelease* in **lp.c**.

Interrupt Routines

Expansion boards have two interrupt levels available, level 1 and level 5 (additional interrupt levels may be made available in the future). Each interrupt level may have more than one device associated with it. All the interrupt service routines at a given level are chained together. When an interrupt occurs, all the routines in the correct chain are executed until one routine returns non-zero. If all routines in the chain return zero, a "Spurious interrupt" message is logged in **/usr/adm/unix.log**.

It is the responsibility of the interrupt routine for a device to return 0 if it is called with no interrupt outstanding on its device and return non-zero and clear the interrupt if one does exist for its device. In this way drivers need only be aware of their own devices and not other devices that may interrupt at the same level.

The routines *set_int()* and *clr_int()* are provided to add and delete interrupt service routines from interrupt chains.

The routine *getslot()* is provided to search for a particular board. *Getslot()* returns the slot address of the board, or 0 if not found. Drivers should use *getslot()* or scan the table **slots**[] which is initialized during boot. Drivers should *not* look in the hardware ID fields of the slots as this may disturb the function of other boards (see **<sys/slot.h>**).

EXAMPLE

```
/* init, release, interrupt service routines */
/* for loadable device xyzzy */
#include <sys/drv.h>
#define XYZ_LEVEL        1          /* interrupt level */
#define XYZ_ID           0x1234     /* board id of xyzzy */
                                    /* expansion board */
#define XYZ_BUSY         1          /* flags */
#define XYZ_OPEN
int xyzzint();                /* interrupt service routine */
struct drv_int xy_int = { 0,xyzzying }; /* struct defined */
                                        /* in drv.h */
int xy_base;
int xy_flags;
xy_init()
{
        if (!(xy_base=getslot(XYZ_ID))
        {
                u.u_error = ENODEV;
                return;
        }
        set_int(&xy_int, XYZ_LEVEL);
        .
        .
        <do hardware initialization>
        .
}
xy_release()
{
        if (xy_flags & (XY_BUSY | XY_OPEN))
        {
                u.u_error = EBUSY;
                return;
        }
        .
        .
        <turn off device>
        .
        clr_int(&xy_int, XYZ_LEVEL);
}
xyzzyint()
{
        if (<not interrupt from xyzzy device>)
                return 0;
        .
        <clear interrupt>
        .
        <process interrupt>
        .
        return 1;
}
```

NAME

error – error-logging and eprintf interface

DESCRIPTION

This device is the interface between the kernel's error-record buffer and user processes. The kernel maintains a ring buffer of records of the following form:

```
struct err
{
        int     e_pid;              /* pid of originator */
        char    e_text[ERRLEN];     /* text of message */
}
```

To read an entry from the queue, a process does a read with enough space to hold one **err** structure. If there is nothing in the queue, the read will block until someone writes to it. On the UNIX PC, a user-level process called the status manager (**/etc/smgr**) generally posts a read on this device and tells the user at the console about any records which appear in the queue.

To write to the queue, a process does a write call supplying a pointer to at most ERRLEN-1 characters (do *not* supply a pointer to an **err** structure). The kernel will fill in the process' pid at the time of the write. A *printf* style interface to the error device is generally available as *eprintf*(3T).

In addition to writes issued by user processes, the kernel sometimes logs hardware errors, etc., in the error log. In this case, the pid returned on the read will be zero. A kernel subroutine called *eprintf* is available for those who wish to write into the error log from kernel code or loadable device drivers.

FILES

/dev/error
/usr/include/sys/err.h
/etc/smgr

SEE ALSO

eprintf(3T).

DIAGNOSTICS

Returns EIO on write of too many characters or read with too little memory to hold **err** structure.

NAME

escape – output escape codes for bitmap windows

DESCRIPTION

This lists the escape sequences honored by the **/dev/window** device (see *window*(7)). *Pn* refers to a numeric parameter (which defaults to 1). *Ps* refers to a selective parameter which defaults to zero.

Cursor Positioning

Name	Sequence	Operation
BEL	\007	Bell (beep); restores screen display if blanked (see scrset(1))
BS	\010	Backspace 1 column. No-op in column 1
HT	\011	Move to next multiple of 8 columns
LF	\012	Down 1 line, scrolling as necessary
VT	\013	See LF (above)
FF	\014	See LF (above)
CR	\015	Cursor to column 1
SO	\016	Shift out (select font 1)
SI	\017	Shift in
NEL	ESC E	Position to column 1 of next line
RI	ESC M	Negative line feed (scroll down at top)
IND	ESC D	See LF (above)
CUU	ESC [Pn A	Cursor up Pn lines
CUD	ESC [Pn B	Cursor down Pn lines
	ESC [Pn e	
CUF	ESC [Pn C	Cursor forward Pn columns
	ESC [Pn a	
CUB	ESC [Pn D	Cursor backward Pn columns
CUP	ESC [Pl ; Pc H	Cursor position to Pl, Pc (1,1 = home)
HVP	ESC [Pl ; Pc f	See CUP (above)

Scrolling, Deleting, Inserting, and Erasing

SU	ESC [Pn S	Scroll entire display up Pn lines
SD	ESC [Pn T	Scroll entire display down Pn lines
DCH	ESC [Pn P	Delete Pn Positions
ICH	ESC [Pn @	Insert Pn Positions
DL	ESC [Pn M	Delete Pn lines
IL	ESC [Pn L	Insert Pn lines
EL	ESC [Ps K	Erase parts of line
EL0	ESC [0 K	Erase cursor to EOL
EL1	ESC [1 K	Erase BOL to cursor
EL2	ESC [2 K	Erase entire line

ED	ESC [Ps J	Erase parts of display
ED0	ESC [0 J	Erase cursor to EOD
ED1	ESC [1 J	Erase BOD to cursor
ED2	ESC [2 J	Erase entire display (clear)

Select Graphic Rendition

SGR	ESC [Ps;Ps;...m	Select graphic rendition (attribute)
SGR0	ESC [0 m	Select normal attribute
SGR1	ESC [1 m	Select bold attribute
SGR2	ESC [2 m	Select dim (dithered) attribute
SGR4	ESC [4 m	Select underline attribute
SGR7	ESC [7 m	Select reverse video attribute
SGR9	ESC [9 m	Select struck-out attribute (ISO)
CTSGR	ESC [= Ps ; Ps m	1st Ps = on mask 2nd Ps = off mask mask = sum of any of: A_UNDERLINE A_REVERSE A_BOLD A_STRIKE A_DIM

Select Character Set

SGR10	ESC [10 m	Select font 0 (see SI)
SGR11	ESC [11 m	Select font 1 (see SO)
SGR12	ESC [12 m	Select font 2
SGR13	ESC [13 m	Select font 3
SGR14	ESC [14 m	Select font 4
SGR15	ESC [15 m	Select font 5
SGR16	ESC [16 m	Select font 6
SGR17	ESC [17 m	Select font 7

Cursor Visibility

CTVIS	ESC [= Ps C	Select cursor visibility and anchoring
CTVIS0	ESC [= 0 C	Normal (cursor on)
CTVIS1	ESC [= 1 C	Invisible (cursor off)

Line Wrap

CTWRAP	ESC [= Ps w	Select line wrap
CTWRAP0	ESC [= 0 w	Wrap off
CTWRAP1	ESC [= 1 w	Wrap on

SEE ALSO

window(7), kbd(7), ANSI Specification X3.64.

NAME

gd – general disk driver

DESCRIPTION

Gd provides the interface to the internal winchester disk and the internal floppy disk.

Eight *ioctl*(2) system calls are available. Two of these use the following structure, defined in **<sys/gdioctl.h>**:

```
struct gdctl
{
        unsigned short    status;   /* Status */
        struct gdswprt    params;   /* Description of the disk*/
        short             dsktyp;   /* The type of disk */
};
```

```
#include <sys/gdioctl.h>
ioctl (fildes, command, arg)
struct gdctl *arg;
```

For additional information on the fields in the **gdctl** structure, refer to **<sys/gdisk.h>**.

The commands are:

GDIOC — Returns the driver ID word. This is a 16 bit quantity where the upper 8 bits are the character 'G' and the 8 low order bits are zero.

GDGETA — Get **gdctl** structure.

GDSETA — Set **gdctl** structure.

GDDISMNT — Dismount the disk. On floppy disk, this also turns off the select light.

GDFORMAT — Format track command. The format track command takes a format buffer pointer as its argument.

Three of the *ioctl* calls are available for the floppy only. One of these, GDCMD, uses the following structure, defined in **<sys/gdioctl.h> :**

```
struct fdrq
{
        char    cmd;   /* Command byte to 2797. */
                       /* Recognized commands are: */
                       /* 100xxxxx      Read sector */
                       /* 101xxxxx      Write sector */
                       /* 1100xxxx      Read address */
                       /* 1110xxxx      Read track */
                       /* 1111xxxx      Write track */
                       /* Fields marked x are not checked and */
                       /* should be filled in by the caller */
                       /* according to information in the */
                       /* WD2797 data sheet. */
        char    cyl;   /* Cylinder to 2797. Note that on a */
                       /* double sided floppy a cylinder has */
```

```
                                /* 2 tracks. Which of these tracks is */
                                /* addressed is determined by the SS0 */
                                /* bit in the command byte.  On return */
                                /* this byte holds the 2797 track */
                                /* register contents after operation */
                                /* completion. */
        char     sec;           /* Sector to 2797. On return this byte */
                                /* holds the 2797 sector register contents */
                                /* after operation completion. */
        ushort   count;         /* Byte count for transfer. The returned */
                                /* count's 14 low order bits hold the 2s */
                                /* complement of the number of words */
                                /* left to do of the DMA transfer. Bits */
                                /* 15 and 14 are indeterminate. This */
                                /* means that for a successful transfer */
                                /* the value should be (xx11)fff. */
        char     stat;          /* Status byte from 2797 */
};
```

```
#include <sys/gdioctl.h>
ioctl (fildes, command, arg)
struct fdrq *arg;
```

The commands are:

GDCMD
: The **fdrq** structure is used by the GDCMD ioctl for "direct" access to the WD2797 by user programs. It sits at the head of the data buffer used for the transfer.

```
#include <sys/gdioctl.h>
ioctl (fildes, command, arg)
char arg;
```

GDRETRY
: Used for turning off/on floppy retries. *arg* is 1 to turn off retries, 0 to turn them back on.

GDLOCK1
: Used to lock out other users from using the floppy disk drive. *arg* 1 = Lock floppy, 0 = Unlock floppy. GDLOCK is implicit when using GDCMD.

FILES

/dev/fp*
/dev/rfp*
/usr/include/sys/gdioctl.h
/usr/include/sys/gdisk.h

NAME

kbd – keyboard codes

DESCRIPTION

The following table gives the sequence of bytes sent for each key pressed on the system console.

Legend gives the keycap legend, *X* gives the sequence sent when that key alone is pressed, *s-X* when shift is pressed, *c-X* when ctrl (control) is pressed with it.

The *Type* field identifies the key type:

SYS = no repeating, no caps lock, no num lock (e.g. Exit)
REPT = repeating, no caps lock, no num lock (e.g. Dlete Char)
ALPHA = repeating, caps lock, no num lock (e.g. A)
NUM = no repeating, no caps lock, num lock (e.g. Home)
NUMREPT = repeating, no caps lock, num lock (e.g. Next)

In the sequences sent, \E means ESC (0x1B), **n* means system special key class *n*. Following the digit *n* is the sequence, thus: *2\EXY means special class 2, send ESC X Y. The classes are established via the WIOCSYS window *ioctl* (see *window*(7)).

ILLK refers to an illegal key combination.

Legend	X	s-X	c-X	Type
Clear Line	\EOa	\EOA	\EOA	SYS
Rstrt/Ref	\EOb	\EOB	\EOB	SYS
F1	\EOc	*1\EOC	ILLK	SYS
F2	\EOd	*1\EOD	ILLK	SYS
F3	\EOe	*1\EOE	ILLK	SYS
F4	\EOf	*1\EOF	ILLK	SYS
F5	\EOg	*1\EOG	ILLK	SYS
F6	\EOh	*1\EOH	ILLK	SYS
F7	\EOi	*1\EOI	ILLK	SYS
F8	\EOj	*1\EOJ	ILLK	SYS
Exit	\EOk	\EOK	\EOK	SYS
Msg	*2 32	*2\032	*2\032	SYS
Help	\EOm	\EOM	\EOM	SYS
Creat	\EOn	\EON	\EON	SYS
Save	\EOo	\EOO	\EOO	SYS
Suspd	*0\EOp	*0\EOP	*0\EOP	SYS
Rsume	*0\EOq	*0\EOQ	*0\EOQ	SYS
Opts	\EOr	\EOR	\EOR	SYS
Undo	\EOs	\EOS	\EOS	SYS
Redo	\EOt	\EOT	\EOT	SYS
Del/Esc	\033	\177	ILLK	REPT
1	1	!	\EPa	REPT
2	2	@	\EPb	REPT
3	3	#	\EPc	REPT
4	4	$	\EPd	REPT
5	5	%	\EPe	REPT
6	6	^	\EPf	REPT

Legend	X	s-X	c-X	Type
7	7	&	\EPg	REPT
8	8	*	\EPh	REPT
9	9	(	\EPi	REPT
0	0	)	\EPj	REPT
-	-	_	\EPk	REPT
=	=	+	\EPl	REPT
Back Space	\010	\010	\010	REPT
Reset/Break	ILLK	\Ec	\Ec	SYS
Cmd	\EOu	\EOU	\EOU	SYS
Close/Open	\EOv	\EOV	\EOV	SYS
Cancl	\EOw	\EOW	\EOW	SYS
Find	\EOx	\EOX	\EOX	SYS
Rplac	\EOy	\EOY	\EOY	SYS
Tab	\011	\EOZ	\EOZ	REPT
Q	q	Q	\021	ALPHA
W	w	W	\027	ALPHA
E	e	E	\005	ALPHA
R	r	R	\022	ALPHA
T	t	T	\024	ALPHA
Y	y	Y	\031	ALPHA
U	u	U	\025	ALPHA
I	i	I	\011	ALPHA
O	o	O	\017	ALPHA
P	p	P	\020	ALPHA
[	[	{	\033	REPT
]	]	}	\035	REPT
\	\	|	\034	REPT
`	`	~	\000	REPT
Print	\EOz	*0\EOZ	*0\EOZ	NUM
Clear/Rfrsh	\ENa	\E[J	\E[J	NUM
Page	\E[U	\E[V	\E[V	NUM
Move	\ENc	\ENC	\ENC	SYS
Copy	\ENd	\END	\END	SYS
Caps Lock	ILLK	ILLK	ILLK	SYS
A	a	A	\001	ALPHA
S	s	S	\023	ALPHA
D	d	D	\004	ALPHA
F	f	F	\006	ALPHA
G	g	G	\007	ALPHA
H	h	H	\010	ALPHA
J	j	J	\012	ALPHA
K	k	K	\013	ALPHA
L	l	L	\014	ALPHA
;	;	:	ILLK	REPT
'	'	"	ILLK	REPT
Return	\015	\015	\015	REPT
Beg	\E9	\ENB	ENB	NUM
Home	\E[H	\ENM	ENM	NUM
End	\EO	\ENN	\ENN	NUM
Dlete	\ENe	\ENE	\ENE	SYS

Legend	X	s-X	c-X	Type
Dlete Char	\ENf	\ENF	\ENF	REPT
Left Shift	ILLK	ILLK	ILLK	SYS
Z	z	Z	\032	ALPHA
X	x	X	\030	ALPHA
C	c	C	\003	ALPHA
V	v	V	\026	ALPHA
B	b	B	\002	ALPHA
N	n	N	\016	ALPHA
M	m	M	\015	ALPHA
,	,	<	ILLK	REPT
.	.	>	ILLK	REPT
/	/	?	ILLK	REPT
Right Shift	ILLK	ILLK	ILLK	SYS
Enter	\012	\012	\012	SYS
Prev	\ENg	\ENG	\ENG	NUMREPT
Roll Up	\E[A	\E[T	\E[T	NUMREPT
Next	\ENH	\ENH	\ENH	NUMREPT
Slect/Mark	\ENi	\ENI	\ENI	SYS
Input Mode	\ENj	\ENJ	\ENJ	SYS
Left Ctrl	ILLK	ILLK	ILLK	SYS
Space	\040	\040	\040	REPT
Right Ctrl	ILLK	ILLK	ILLK	SYS
Num Lock	ILLK	ILLK	ILLK	SYS
<-	\E[D	\ENK	\ENK	NUMREPT
Roll Down	\E[B	\E[S	\E[S	NUMREPT
->	\E[C	\ENL	\ENL	NUMREPT

NAME

ktune – kernel tunable parameters

SYNOPSIS

ktune [options] [files]
ktune [**–d**] [**nbuf**=n] [**inode**=n] [**nfile**=n]
[**nproc**=n] [**ntext**=n] [**nclist**=n] [**npbuf**=n]
[**ncall**=n] [**nttyhog**=n] [**kern**=filename]

DESCRIPTION

Ktune provides a way to change values of the following parameters which reside in the file 'filename' specified in the argument 'kern=filename'. If the argument 'kern=filename' is absent, the program modifies **/unix**.

The table below summarizes the parameters that can be set using *ktune*.

Parameter	**Minimum Value**	**Default**
nbuf	25	100
ninode	80	400
nfile	80	300
nproc	30	100
ntext	24	75
nclist	32	150
npbuf	4	16
ncall	16	32
nttyhog	0	1024

nbuf Number of system buffers available. These buffers are used mostly by block device drivers for file system operations.
Range: 25 up to system capacity.

ninode Number of memory-resident inodes that can be allocated at any time. The inode is the focus of all file activity in UNIX. There is a unique inode allocated for each open file, each current directory, each mounted-on file, text file, and the root.
Range: 80 up to system capacity.

nfile Total number of files that can be opened on the system at any time. One file structure is allocated for each open/creat/pipe call. Note that while **nfile** controls the *total* number of files that can be open at any given time, another parameter, **nopen**, sets the number of files that can be open at any given time by any *single* process. **Nopen** is not tunable, and is currently set to 80.
Range: 80 up to system capacity.

nproc Number of processes that can exist at any time. One process structure is allocated per active process, and it contains all the data about the process.
Range: 30 up to system capacity.

ntext Number of text structures allocated in the kernel. One text structure is allocated per pure procedure on swap devices.
Range: 24 up to system capacity.

nclist Number of clist buffers available. These buffers are used mostly by character device drivers for terminal I/O operations.
Range: 32 up to system capacity.

npbuf Number of buffer headers available in the raw I/O pool of headers.
Range: 4 up to system capacity.

ncall Number of callouts allowed in the kernel. When a process must be sure that it is awakened after a specific period of time, it calls the kernel timeout routine with a specified amount of time. The timeout routine places an entry in the callout table. **ncall** specifies the number of entries in the callout table.
Range: 16 up to system capacity.

nttyhog Maximum number of characters outstanding in the tty buffer for a given port before the system will flush that port's queue. If this value is set to 0, the system will no longer check for the maximum characters outstanding in the buffers. The **ttyhog** option keeps one port from using all the clist buffers, ensuring that each port has enough buffer space.
Range: 0 up to 1024

kern= If an argument 'kern=filename' is present, the program modifies file 'filename' instead of **/unix**.

ktune commands that list only some of the parameters cause only those parameters to change. An argument consisting of a dash (−) is taken to be a file name corresponding to the standard input. The *options* may appear in any order but must appear before the *files.*

The **−d** flag lists each parameter and the value which the kernel is currently using. Note that this might be different than the setting on the actual file on disk. Each parameter appears on a separate line, with the value preceded by a keyword (i.e., **ninode**=200). Input lists that list only some of the parameters cause only those parameters to change. This option displays the actual settings in use for the running kernel (not the settings stored on the disk). These settings may be lower than the disk settings due to small memory size.

There is a table called 'tuhi' which resides in the kernel. Tuning is accomplished by changing the parameters in this table on the disk, and **requires** the user to **reboot**.

All input parameters are checked against a set of minimum parameters. Any input with an error on any parameter results in no changes to any parameters. Input lists containing a value that violates these minimums result in no changes, and an error return.

The kernel boot routine is modified to provide for sanity checking on boot up to insure that enough memory is present for the values specified, and that the kernel virtual memory addressing limits are not violated. If the memory found is too small for the values in 'tuhi', the values in core (not on the disk) are adjusted downward until the resulting kernel runs on the system being booted. If after ten refinements the values in 'tuhi' are still too large, the default tuning is used.

NAME

lp – line printer
rawlp – raw line printer

DESCRIPTION

Lp provides the interface to any standard Centronics line printer. When it is opened or closed, a suitable number of page ejects are generated. Bytes written are printed.

An internal parameter within the driver determines whether or not the device is treated as having a 96- or 64-character set. In half-ASCII mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

```
{    -(
}    -)
`    -'
|    -!
~    -^
```

The driver correctly interprets carriage returns, backspaces, tabs, and form-feeds. A new-line that extends over the end of a page is turned into a form-feed. The default line length is 132 characters, no indentation and lines per page is 66. Lines longer than the line length minus the indent (i.e. 132 characters, using the above defaults) are truncated.

Two *ioctl*(2) system calls are available:

```
#include <sys/lprio.h>
ioctl (fildes, command, arg)
struct lprio *arg;
```

The *commands* are:

LPRGET Get the current indent, columns per line, and lines per page and store in the *lprio* structure referenced by **arg**.

LPRSET Set the current indent, columns per line, and lines per page from the structure referenced by **arg**.

Thus, indent, page width and page length can be set with an external program.

Rawlp provides a direct interface to the parallel printer with no modification of the data sent.

FILES

/dev/lp
/dev/rawlp

SEE ALSO

lp(1).

NAME

mem, kmem – core memory

DESCRIPTION

Mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system.

Byte addresses in *mem* are interpreted as memory addresses. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

On the PDP-11, the I/O page begins at location 0160000 of *kmem* and per-process data for the current process begins at 0140000.

FILES

/dev/mem, /dev/kmem.

NAME

null – the null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

phone – telephony interface and control

DESCRIPTION

The telephone ports support both voice and data functions. Depending on the operating mode, the lines behave quite differently. The remainder of this discussion describes the proper use of the various voice and data features of this device.

There are two ports available for general use. Each port can operate in either voice or data mode, though switching between the two modes on the same call is not allowed. Only one port can operate in data mode at any time, since there is only one data element (Bell 212 compatible) shared between the two ports. The ports can operate in voice mode simultaneously; the handset is shared on an as-needed basis.

Phone lines are opened and closed via *open*(2) and *close*(2). Programs access the phone lines by opening the file:

/dev/ph0	Device filename for Line 1.
/dev/ph1	Device filename for Line 2.

Opening the line determines the operating mode of the phone lines until the line is closed. The modes are:

open ("/dev/ph1", O_RDWR | O_NDELAY);
: Open the line for outgoing data call. The open will return immediately for dialing.

open ("/dev/ph1", O_RDWR);
: Open the line for incoming data call. The open will not return until an incoming data call has been received and connection established.

open("/dev/ph0", O_RDONLY)"
: Open the line for voice calls. The open will return immediately ready to make or receive voice calls.

The easiest way to establish connection for outgoing data calls is via the *dial* and *undial* library routines (see *dial*(3C). This should be the preferred method for using the ports because the library routine has been modified to maintain lock files, and purge and restore any *getty*(1M) whenever the phone or serial ports are used.

In data mode, the port behaves much like a standard asynchronous port, using *read*(2) and *write*(2) to perform I/O on the phone line. In addition to the line parameters and *ioctl* commands described in *termio*(7), the phone device provides additional *ioctl* commands which control and monitor telephony operation. The following structure is used primarily for this function, defined in **<phone.h>**:

```
struct updata {
        char        c_lineparam;          /* line params */
        char        c_waitdialtone;       /* timeout value */
        ushort      c_linestatus;         /* line status */
        ushort      c_feedback;           /* ring control */
        ushort      c_waitflash;          /* flash period */
};
```

The *c_lineparam* field describes the functions of the phone lines:

VOICE	001	Line used for voice (read only).
DATA	002	Line used for data (read only).
DTMF	004	Use digitone for dialing.
PULSE	010	Use pulse for dialing.
INCMNG	020	Answer incoming calls.
MSGWAIT	040	Detect Message Waiting
USEALEAD	100	Use A-lead.

VOICE and DATA indicate which mode the line is currently set to. These are read only and do not change after the initial open.

If DTMF is set, the line will use touch-tone for dialing.

If PULSE is set, the line will use pulse code for dialing.

If INCMNG is set, when ringing is detected on the line, the call will be answered and a data call connection is attempted. If a connection is established, the process is notified through either a *wakeup()* or *signal()*.

If MSGWAIT is set (valid for line 1 only), whenever the Message Waiting signal changes from absent to present or from present to absent, a *signal*(2) is sent to notify the process of the change.

If USEALEAD is set (most keyset lines), the A-lead is used for all call functions.

When a line is opened in the data mode, the default parameters set are DATA and INCMNG. For outgoing data calls, reset the INCMNG.

The *c_waitdialtone* field specifies timeout value, in units of seconds, for dialtone detection. If tone isn't detected within this interval, an error is returned. The default value is set to 5 seconds.

The *c_linestatus* field reflects the current state of the line and should be of interest only to lines which operate in voice mode. If a voice line changes state, this field is modified and a signal (SIGPHONE) is sent to the process. In the signal catching routine, the process should perform a PIOCGETP to read the current status and compare it with the previous status for changes.

MESSAGE	0000001	Message waiting detected. This bit is valid only if MSGWAIT is set. This bit is set when Message Waiting is detected. A signal is sent to the process whenever this bit is modified.

SETOFFHOOK	0000002	Handset is lifted off the cradle. This bit is set for the line connected to the handset whenever the set is lifted. A signal is sent to the process whenever this bit is modified.
INCOMERING	0000003	Ringing is detected on the line. This bit follows the ring signal on the line and therefore a signal is sent to the process on each transition.
MODEMCONNECTED	0040000	Modem handshake complete. This bit is set when the modems on both ends are synchronized for data transmission. No signals are sent when this bit is updated.

The *c_feedback* field controls the various functions of the onboard dialer for feedback purposes:

SPEAKERON	0000001	Setting this bit allows the user to monitor the call through the onboard speaker.
SOFTSPK	0000002	Speaker volume control for call monitoring.
NORMSPK	0000004	
LOUDSPK	0000006	
RINGON	0000020	Setting this bit causes ringing to be generated on the onboard speaker instead of the handset.
SOFTRING	0170000	Ringer volume control for incoming calls.
NORMRNG	0070000	
LOUDRNG	0150000	
LOWRNG	0000000	Ringer pitch control for incoming calls.
MEDRNG	0004000	
HIMEDRNG	0010000	
HIRNG	0020000	

The *c_waitflash* field specifies the amount of time, in units of milliseconds, that the hook switch will remain closed during the *ioctl* for hook flash.

The *ioctl*(2) system calls are used to set and read the status of the phone line. They have the form:

```
ioctl( filedes, command, arg)
struct update *arg;
```

The commands using this form are:

PIOGETP Get the parameters and status associated with the phone line and store in the *updata* structure referenced by *arg*.

PIOSETP Set the parameters associated with the phone from the structure referenced by *arg*. The read-only portion of the structure is ignored.

Another *ioctl*(2) call has the form:

```
ioctl( filedes, command, arg)
char *arg;
```

The commands using this form are as follows:

PIOCDIAL Dial the digit or perform the function associated with the character.

Digits "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "#", and "*" are dialed. The "#" and "*" characters are ignored when dialing in digit pulse mode.

Characters "~", "=", and "+" are used equivalently for pausing for the next tone. If tone is not detected within the specified timeout period, an error is returned.

Character "," causes a pause of 2 seconds.

Character ":" causes a pause of 10 seconds.

Characters "&" and "!" are used equivalently for performing a hook switch flash operation.

Character "$" is used to terminate the dialing sequence. If the line is used for data, this call will not return until either a connection is established or until the automatic timeout of 32 seconds is reached. In the latter case, an error is returned.

Character "@" can be used to terminate the dialing sequence for a data line. The call returns immediately, the program then periodically reads the status of the line (MODEMCONNECTED of *c_linestatus*) to determine if the connection has been established. There is no timeout limitation in this mode.

Character "%" begins Touch-tone dialing from that point in the dial string.

Character "^" begins pulse dialing from that point in the dial string.

Characters "/ /" are used for alphabetic dialing. Letters typed between the slashes are translated to the corresponding numbers on the dial pad.

All other characters are ignored.

Additional *ioctl*(2) calls for controlling phone line functions:

PIOCOFFHOOK — This should be the first *ioctl*(2) call to set up the line for dialing. Error will be returned if resources necessary for the call are not available.

PIOCDISC — Terminate the call and release resources.

PIOCHOLD — Put the line on hold.

PIOCUNHOLD — Reconnect the call to the handset.

PIOCRECONN — Reconnect the dialer for more dialing.

PIOCFLASH — Perform a hook flash for the specified interval. This is used after dialing is completed.

PIOC1800 — Toggles bit 5 of register 5 (b5R5). If b5R5 = 1 then an 1807 Hz. guard tone is transmitted by the answer mode modem during data transmission. Setting this bit to 0 disables the 1807 Hz. guard tone (212A compatible).

PIOCANSTONE — Toggles bit 0 of register 4 (b0R4). If b0R4 = 1 then the answer tone is set to be PSK unscrambled mark (CCITT compatible). If b0R4 = 0 then the answer tone is set to be a 2225 Hz. FSK tone (212A compatible).

PIOC2100 — Toggles bit 6 of register 5 (b6R5). If b6R5 = 1 then the 2100 Hz. European answer tone is set to precede the handshaking sequence of the answer mode modem (CCITT compatible) If b6R5 = 0 then a 212A compatible handshaking sequence is selected (no 2100 Hz. tone).

PIOCOVRSPD — Toggles bit 6 of register 4 (b6R4). If b6R4 = 1 then character overspeed of 2.3% over nominal bit rate will be applied in asynchronous mode. If b6R4 = 0 then overspeed of 1% (212A compatible) will be used.

PIOCDSRAFT — Toggles bit 7 of register 5 (b7R5). If b7R5 = 0 then DSR (b4R2) of the answer mode modem is turned on after the silent interval following the 2100 Hz. tone (CCITT compatible). If b7R5 = 0 then DSR of the answer mode modem is turned on when the modem is connected to the line (212A compatible).

Applications can be interfaced to the phone manager using the **/usr/lib/ua/Comm_pkgs** file. The table below summarizes the actions taken by the phone manager when an application is invoked.

Port and Device Type: Phone Manager Actions

Phone	**Port**	**Device**	**Phone Manager Action**			
Number	**Type**	**Type**	**Set-up**	**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 PHMGR takes one of six paths. **Port types** are either *serial* or *ph* (phone). **Device types** are either *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.

FILES

/dev/ph*

SEE ALSO

phonedvr(7), termio(7), ioctl(2), open(2), ua(4).

NAME

phonedvr – Kernel structure interface and control

DESCRIPTION

Phonedvr provides procedures for developers who want to write their own phone driver.

Four procedures provide the timing needed by communications applications. These are described below.

int add_hi_scan (func)
***func**

This procedure takes a pointer to a function as the input argument and returns 0 if the add operation is successful or −1 if the table is completely full and no additional arguments may be added. The added function will be called sixty times a second (60MH) and should be used when critical sections of code need to be executed without interruption. Extreme care must be taken when using this feature since *system interrupts are disabled* when the function is executing. Otherwise, the rest of the system will be locked out for long periods of time. One use of this feature would be to do accurate timing for pulse dialing.

int rm_hi_scan (func)
***func**

This procedure takes a pointer to a function as the input argument and returns 0 if the function is found and removed from the system table, or −1 if the function is not found.

int add_lo_scan (func)
***func**

This procedure takes a pointer to a function as the input argument and returns 0 if the add operation is successful or −1 if the table is completely full and no additional arguments may be added. This function is used for less critical sections of code that need to be executed sixty times a second (60HZ). It is executed with interrupts enabled. Functions that can tolerate being interrupted during execution should use this feature.

int rm_lo_scan (func)
***func**

This procedure takes a pointer to a function as the input argument and returns 0 if the function is found and removed from the system table, or −1 if the function was not found. This is the complement to the **add_lo_scan** procedure.

The above routine is useful for adding or removing functions when writing loadable device drivers. It is an external function.

SEE ALSO

phone(7) termio(7), ioctl(2), open(2), ua(4).

NAME

qt – QIC-II streaming tape driver

DESCRIPTION

The *qt* loadable device driver provides the interface to one QIC-II streaming tape drive via a QIC-II controller on an expansion board.

Four *ioctl*(2) system calls are available. They use the following structure, defined in **<sys/qtioctl.h>**:

```
struct qtio
{
        unsigned short    status[6]; /* Tape drive status */
        int               bcnt;      /* Number of 512 byte blocks */
                                     /* transferred since last open */
};
```

```
#include <sys/qtioctl.h>
ioctl (fildes, command, arg)
struct qtio *arg;
```

The commands are:

QTIOC Returns the driver ID word. This is a 16 bit quantity where the upper 8 bits are the character 'Q' and the 8 low order bits are the minor device number.

QTGETA Get *qtio* structure.

QTSETA Set *qtio* structure.

QTCMD Send auxilliary tape command.

QTCMD arguments:

ERASE	6	Erase tape.
RETEN	7	Retension tape.

EXAMPLES

To put a single *cpio* save set on a tape, do the following:

```
</dev/rmt0
```

to rewind tape;

```
find . -print | cpio -ocvT >/dev/rmt0
```

to backup from current path.

To put several *cpio* sets on one tape, use **/dev/rmt4**. To skip to the next save set when reading, do:

```
</dev/rmt4
```

Note that data can only be written from the very beginning of the tape, or appended to the end after the last save set. If the tape is at the beginning, writing will occur from the beginning, overwiting whatever may be on the tape. If the tape is not at the beginning, writing will start after the last valid data on the tape.

Also note that the *cpio* **–T** option should be used whenever possible to avoid unneccesary tape wear. If the tape was written using the **–B** option you must use that option on read. or an error may occur towards the end of the save set.

FILES

/dev/rmt0	This device rewinds the tape on close.
/dev/rmt4	This device positions the tape at the next file mark on close.

NAME

termio – general terminal interface

DESCRIPTION

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character # erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character @ kills (deletes) the entire input line, and optionally outputs a new-line character. Both of these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).

QUIT (Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF (Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (ASCII NUL) is an additional line delimiter, like NL. It is not normally used.

STOP (Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hangup* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in <**termio.h**>:

```
#define NCC      8
struct   termio
{
         unsigned  short  c_iflag;     /* input modes */
         unsigned  short  c_oflag;     /* output modes */
         unsigned  short  c_cflag;     /* control modes */
         unsigned  short  c_lflag;     /* local modes */
         char             c_line;      /* line discipline */
         unsigned  char   c_cc[NCC];   /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	INTR	DEL
1	QUIT	FS
2	ERASE	BS
3	KILL	@
4	EOF	EOT
5	EOL	NUL
6	reserved	
7	reserved	

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7 bits, otherwise all 8 bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.

BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2 four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
B19200	0000016	19200 baud
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.
CTSCD	0010000	Use hardware flow control.
HDX	0020000	Set line in half-duplex mode.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

If CTSCD is set, flow control is performed using hardware signals. No data will be sent in the absence of the CTS (Clear To Send) signal. Outgoing data will be suspended if CTS is lowered, transmission will resume after CTS is raised.

If HDX is set, the RTS (Request To Send) signal will not be raised until an *ioctl* command is issued. If a write is attempted before CTS is present, an error will be returned.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g. 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
‘	\’
|	\!
~	\^
{	\(
}	\)
\	\\

For example, **A** is input as **a**, **n** as **n**, and **N** as **n**.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.

TCSETA Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.

TCSETAW Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.
TCSRTS	If *arg* is 0, turn off RTS; if 1 turn on RTS. Error will be returned if CTS is not present within 1 second of turning on RTS. This command should be used on lines that operate in half-duplex mode.

FILES

/dev/tty*

SEE ALSO

stty(1), ioctl(2), window(7).

NAME

tty – controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal or window associated with the process group of that process, if any. It is useful for programs or shell sequences that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

FILES

/dev/tty
/dev/tty*

SEE ALSO

termio(7), window(7).

NAME

window – bitmap windows

DESCRIPTION

Windows are opened and closed via *open*(2) and *close*(2). To open a new window, the program opens the device **/dev/window**. The kernel will bind a new window to the returned file descriptor. The window number can be obtained in the minor device field of a subsequent *stat*(2) call on the file descriptor.

The opening of a window creates a dimensionless window which does not occupy any screen space. The window size is subsequently established in one of two ways:

Implicitly. If the program does a *read*(2), *write*(2), or certain *ioctl*(2) calls on the window, the kernel will automatically set the window size to a default (currently full-screen) and then proceed with the particular system call.

Explicitly. If the program does an *ioctl*(wd,WIOCSETD,&params), then the window size is taken from the params given (see WIOCSETD, below). This is the preferred mechanism for establishing a window's size as it permits the creation of windows of arbitrary dimensions.

In addition, a program may open **/dev/w***N* where *N* is the window number (minor device number) of an already-existing window. This permits multiple applications to open the same window.

When a window is created, it automatically becomes the current, active window. As soon as dimensions are established, it will be displayed at the front of the screen, unobscured by any other windows. In addition, the default system font is loaded into font slot 0.

Any window may be closed via the *close*(2) system call. When the last of potentially multiple programs closes the window, its space on the display is removed.

Read and *write* calls are used to perform I/O on the window. *Read* reads characters from the keyboard and returns them to the process. *Write* writes characters to the display.

The data to be read or written is in ANSI X3.64 7-bit ASCII code. Output sequences exist to control cursor motion, insertion, deletion, erasure, fonts, and various mode settings. Input consists of characters and control sequences.

In addition, all the standard facilities of the UNIX *tty* driver are available, control of echo, raw mode, new-line, padding, interrupt/quit/kill characters, etc.

Any uncovered window can write to the display without blocking, regardless of whether or not it is the active window. For example, this allows status processes to output system update messages even though they are not the current window.

Reading does not explicitly block non-active windows. Rather, they are allowed to read any input data which was accumulated (typed ahead) when they were last active. When this data is exhausted, the process will block on the next read.

In addition to all *tty ioctls* (TIOCxxxx), the window device supports its own *ioctls* which control window functions:

```
ioctl(wd,WIOCGETD,&uwdata)
ioctl(wd,WIOCSETD,&uwdata)
```

These calls allow the program to get and set (respectively) parameters about the window. The *uwdata* structure has the following form:

```
struct  uwdata                          /* user window */
{
        unsigned short  uw_x;           /* upper-left-corner x */
        unsigned short  uw_y;           /* upper-left-corner y */
        unsigned short  uw_width;       /* width (pixels) */
        unsigned short  uw_height;      /* height (pixels) */
        unsigned short  uw_uflags;      /* various flags */
        unsigned char   uw_hs;          /* horizontal size (RO) */
        unsigned char   uw_vs;          /* vertical size (RO) */
        unsigned char   uw_baseline;    /* baseline (RO) */
};
```

uw_x and *uw_y* are the pixel coordinates of the upper left-hand corner of the window. If the window has borders (see below), then *uw_x* and *uw_y* specify the upper left-hand corner of the border. *uw_width* and *uw_height* are the width and height of the window (in pixels). The width and height never include the border.

uw_vs is the vertical spacing for characters in pixels. *uw_hs* is the horizontal spacing for characters. *uw_baseline* is the vertical offset from 0,0 to the baseline position of a character. The baseline is an imaginary line around which characters are drawn – much like the ruling on a conventional lined note pad.

uw_vs, *uw_hs*, and *uw_baseline* are read-only parameters (they are ignored on a WIOCSETD call). They are computed dynamically by the kernel based on the most extreme character loaded into any font in the window's palette (see WIOCLFONT). All character-oriented cursor motion uses these values to translate character addresses to pixel addresses. In addition, if the VCWIDTH flag is off in *uw_uflags* (see below), all characters are displayed in an imaginary cell which is *uw_hs* X *uw_vs* pixels. In this mode, existing UNIX programs (such as *vi*) can readily address the display as a "normal" character terminal, even if multiple "fancy" proportional-space fonts are used instead of the nominal 9 X 12 system font. Smarter applications use pixel addressing for glyph placement and set the VCWIDTH flag enabling proportional glyph placement.

In addition, *uw_hs* and *uw_vs* control the size of the character cursor in the window. In the future, applications will have far more control over the appearance and size of both the character and mouse cursor.

The *uw_uflags* field contains flags:

```
#define NBORDER      0x1    /* borderless */
#define VCWIDTH      0x2    /* variable chr spacing */
#define BORDHSCROLL  0x4    /* border hscroll icons */
#define BORDVSCROLL  0x8    /* border vscroll icons */
#define BORDHELP     0x10   /* border help patch */
#define BORDCANCEL   0x20   /* border cancel patch */
#define BORDRESIZE   0x40   /* border re-size patch */
#define BORDWNUM     0x80   /* border window num */
#define UNCOVERED    0x100  /* uncovered (RO) */
#define KBDWIN       0x200  /* keyboard (RO) */
#define NOCLEAR      0x400  /* don't clear window */
```

NBORDER turns off the window's borders by forcing the four margin parameters to zero. This bit is vestigial and will probably be deleted shortly. Callers should explicitly zero the four margin parameters to eliminate window borders.

VCWIDTH, when set, enables proportional character placement. When set, the cursor is advanced by the displayed character's horizontal increment, rather than the window-wide maximum character width (readable as *uw_hs*).

The BORDxxxx flags enable the corresponding border icons. HSCROLL enables the horizontal scrolling icons, VSCROLL the vertical. HELP, CANCEL and RESIZE enable the help, cancel and window re-sizing icons. When the mouse is clicked on an enabled icon, the corresponding keyboard sequence is transmitted.

UNCOVERED is set whenever the window is totally visible. This flag is read-only.

KBDWIN is set when this window is the current (keyboard) window. This flag is read-only.

Setting NOCLEAR prevents the system from clearing out the contents of the window upon its creation. This flag is intended for use only by window management system software.

ioctl(**wd,WIOCSELECT**)

This *ioctl* causes the window *wd* to become the current keyboard (active) window. This call is normally issued only by window management software, not by applications.

ioctl(**wd,WIOCREAD,&pixmap**)

This *ioctl* causes the pixel image of the entire display to be "dumped" into the memory at *pixmap*. *Pixmap* should be 15660 unsigned shorts arranged as 348 rows each containing 45 unsigned shorts. The least significant bit of the first short in the array contains the upper-left-hand display pixel.

ioctl(wd,WIOCSETTEXT,&ut)
ioctl(wd,WIOCGETTEXT,&ut)

These *ioctls* allows the application to associate textual data with a window's two screen-labeled key (SLK) lines, command line, and prompt line. These four lines are the bottom four on the display and switch with the selected window. In addition, the application may program the window's label line (the top border) and a non-displayed "user" line which is generally used to describe the window to window management software.

The *ut* (user text) structure has the following form:

```
struct  utdata                          /* user text data */
{
        short   ut_num;                 /* number (see above) */
        char    ut_text[WTXTLEN];       /* text */
};
```

ut_num is the text item number (WTXTSLK1, WTXTSLK2, WTXTPROMT, WTXTCMD, WTXTLABEL, WTXTUSER) and *ut_text* contains the null-terminated data.

ioctl(wd,WIOCSYS,num)
ioctl(wd,WIOCGSYS,num)

The WIOCSYS *ioctl* declares window *wd* to be system window number *num*. WIOCGSYS returns the process group associated with an existing system window number *num*. There are currently three system windows. Each system window "owns" a number of keys on the keyboard, regardless of the currently selected window. If one of these keys is struck, it is queued for reads in the appropriate system window and is not sent to the currently active one. No other action is taken (specifically, the system window is not selected). The following table lists the special system keys:

SYSWMGR(0) Window Manager:
Suspd, s-Suspd, Rsume, s-Rsume, s-Print

SYSPMGR(1) Phone Manager:
All shifted function keys (F1-F8)

SYSSMGR(2) Status Manager:
Msg, s-Msg

ioctl(wd,WIOCGETMOUSE,&umdata)
ioctl(wd,WIOCSETMOUSE,&umdata)

These *ioctls* control the mouse. Once enabled, the mouse sends "reports" to the application in the same stream as keyboard input. If the mouse has not been enabled, no reports are sent.

The *umdata* structure is as follows:

```
#define MSDOWN   0x1   /* buttons go down */
#define MSUP     0x2   /* buttons go up */
#define MSIN     0x4   /* mouse is in rectangle */
#define MSOUT    0x8   /* mouse is outside rect */

struct  umdata                  /* mouse data */
```

```
{
        char            um_flags;    /* wakeup flags */
        short           um_x;        /* motion rectangle */
        short           um_y;
        short           um_w;
        short           um_h;
        struct icon     *um_icon;    /* ptr to icon */
};
```

The *um_flags* field contains flags which are used to determine when mouse reports should be sent. MSUP and MSDOWN cause reports to be sent when buttons go up or down, respectively. MSIN and MSOUT cause reports to be sent when the mouse is located within (MSIN) or outside (MSOUT) the rectangular region specified with *um_x*, *um_y*, *um_w*, and *um_h* (x, y, width, and height, in pixels).

um_icon is an optional pointer to an icon structure (see *font*(4)). This icon will be used as the mouse-track cursor. If *um_icon* is zero, the standard system mouse-track is used.

Mouse reports take the form:

ESC [? {x-pos} ; {y-pos} ; {buttons} ; {reason} M

Where ESC is the ASCII escape character (\033) followed by left square-bracket, question mark and four ASCII decimal numbers separated by semicolons. The sequence is terminated by a capital *M* character.

{x-pos} and {y-pos} are the x and y positions of the mouse-track relative to the window. {buttons} is a single digit character in the range **0** (\060) to **7** (\067) representing three mouse buttons as bits. The most significant bit is the left-most mouse button.

{reason} is an ASCII decimal string explaining what event caused the mouse report. The number consists of combinations of the MSUP, MSDOWN, MSIN, and MSOUT bits (above). Whenever a mouse report is generated due to MSIN or MSOUT, the enable bit for the condition is clear. These wakeup conditions are one-shot. Whenever a WIOCSETMOUSE *ioctl* is issued with the MSIN or MSOUT bits set in *um_flags*, a check is made to see whether an immediate report is necessary because the mouse already satisfies the wakeup condition.

Some typical mouse reports are:

ESC [? 100 ; 20 ; 1 ; 1 M

The reason is MSDOWN (1), the button state is 1 (rightmost button down, others up). The mouse-track is at 100,20.

ESC [? 10 ; 54 ; 0 ; 4 M
The reason is MSIN (4), there are no buttons down (0), the mouse-track is at 10,54 which is within the bounds defined by the rectangle in the last WIOCSETMOUSE *ioctl*.

ioctl(**wd,WIOCRASTOP,&urdata**)
The WIOCRASTOP *ioctl* provides user programs with direct access to a window's pixel data. This "raster operation" *ioctl* is controlled by the *urdata* structure:

```
struct   urdata                              /* user rastop data */
{
         unsigned short  *ur_srcbase;  /* ptr to source data */
         unsigned short  ur_srcwidth;  /* number bytes/row */
         unsigned short  *ur_dstbase;  /* ptr to dest data */
         unsigned short  ur_dstwidth;  /* number bytes/row */
         unsigned short  ur_srcx;      /* source x */
         unsigned short  ur_srcy;      /* source y */
         unsigned short  ur_dstx;      /* destination x */
         unsigned short  ur_dsty;      /* destination y */
         unsigned short  ur_width;     /* width */
         unsigned short  ur_height;    /* height */
         char            ur_srcop;     /* source operation */
         char            ur_dstop;     /* destination operation */
         unsigned short  *ur_pattern;  /* pattern pointer */
};

/* 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 */
```

The first four members of the structure 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 *ioctl*(wd). 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 next four members of the *urdata* structure 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 identify 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.

Since WIOCRASTOP is really an output operation, the process is blocked until the window is exposed. In addition, the raster operation waits for previously-output characters to appear on the screen before commencing.

ioctl(**wd,WIOCLFONT,&ufdata**)
ioctl(**wd,WIOCUFONT,&ufdata**)
ioctl(**wd,WIOCGFONT,&itable**)

WIOCLFONT and WIOCUFONT control the loading and unloading of fonts for a particular window. Each window has 8 font "slots" which are addressable with ANSI X3.64 character strings (SGR, SI, SO, SS2, etc; see *escape*(7) for details of these sequences). Two calls support installable fonts, SYSL_LFONT and SYSL_UFONT; see *syslocal*(2) for details. The WIOCLFONT call loads a font into a slot, automatically unloading any font previously loaded there. WIOCUFONT explicitly unloads a font from a slot. The WIOCGFONT call gets the inode number of fonts currently loaded. The entry in *itable*[] is NULL (0) for unassigned slots, including slot 0 if no font has been explicitly assigned there. Loaded fonts tie up system resources (although the kernel will automatically "share" identical fonts across multiple windows) so it is good practice to unload fonts when they are no longer needed. Note that the font in slot #0 is known as the "system font." and is called upon to produce window text messages and SLK labels. If the font file is malformed, a −1 is returned from the *ioctl* and *errno* is set to EBFONT.

The *ufdata* structure is very simple:

```
#define FNSIZE  60                      /* font name size */

struct  ufdata                          /* user font data */
{
        short  uf_slot;                 /* slot number */
        char   uf_name[FNSIZE];         /* font name (file name) */
};
```

uf_slot is the font slot number (0-7) and *uf_name* is the path name where a suitably-formatted font file can be found. See *font*(4) for more information about fonts.

The *itable* structure is of the form:

```
int itable[8]
```

Each element of the array contains the inode number of the font in the equivalent slot number. Unassigned slots are NULL (0).

When a new font is loaded, the kernel checks to see if it contains any character more extreme than the one reflected in the current *uw_hs*, *uw_vs*, and *uw_baseline* variables. If it does, the three values are updated. When a font is unloaded, the kernel computes new values for *uw_hs*, *uw_vs*, and *uw_baseline*.

ioctl(**wd,WIOCPGRP,dummy)**

This *ioctl* sets the window's controlling process group to that of the process issuing the *ioctl*. It is especially useful in those cases where the parent has *open*ed a new window which it wishes to give to the child. If the child does a *setpgrp*(2) call, it will be isolated from the parent's process group. If the child does not issue this *ioctl*, the signals generated by interacting with the child (e.g. SIGINT) will go to the process group that opened the window (the parent), but the child will not see these signals because it has done the *setpgrp*(2) call.

The recommended sequence is:

```
open the window
fork()

        CHILD:
        setpgrp()
        dups and closes for stdin, out, err
        ioctl(0,WIOCPRGP)
```

exec

PARENT:
wait(), etc.

ioctl **(wd,WIOCGCURR,dummy)**

This *ioctl* returns the window number of the currently selected window. If no window is selected, ENXIO is returned.

ioctl(**wd,WIOCGPREV,dummy)**

This *ioctl* returns the window number of the previously selected window. If no window was selected, ENXIO is returned.

ioctl(**wd,WIOCSCR,num**)

This *ioctl* sets the delay value, in seconds, before the screen will dim. The delay takes effect from the last key hit on the keyboard, or the last time the mouse is touched. If *num* is 0, the screen save feature is disabled. Any positive integer will set the delay to that value. A negative value will dim the screen without changing the value of the delay. This call always returns the previous value of the delay.

FILES

/dev/window*
/usr/include/sys/window.h

SEE ALSO

termio(7), font(4), tam(3T), wrastop(3T), syslocal(2)

BUGS

The VCWIDTH of uw -uflags structure is inoperative on the UNIX PC.

Installing the AT&T UNIX® PC Curses/Terminfo Programmer's Package

The Curses/Terminfo Programmer's Package is included with Version 3.5 of the AT&T UNIX PC UNIX Utilities and runs on the UNIX PC Version 3.5 system software. This package complies with the UNIX System V Operating System V Interface Definition (SVID) and therefore allows developers to take applications from the AT&T 3B2 computer and port, without making any code changes, to the UNIX PC. As in previous versions of UNIX PC software, the curses library cannot be used with TAM or shared libraries.

This package consists of one disk labeled "Curses/Terminfo Programmer's Disk."

To install the disk:

1. From the Office of install, open |**Administration**|.

   You see the Administration menu.

2. Select |**Software Setup**| and press <**Enter**>.

   You see the Software window.

3. Select |**Install Software from Floppy**| and press <**Enter**>.

4. Insert the disk and press <**Enter**>.

   You see a window asking you to insert the floppy disk.

   Shortly there after you see the message: Install in progress on your screen.

5. You are notified when to remove the floppy disk and when the installation is complete.

6. When the installation is complete, close the Software windows.

Note: If you remove this package, the previous version of curses will be restored.

Appendix

Curses/Terminfo Programmer's Guide

PAGE

Appendix

Curses/Terminfo Programmer's Guide

Introduction

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the Terminal Information Utilities package, commonly called **curses/terminfo**, to write screen management programs on a UNIX system. This package includes a library of C routines, a database, and a set of UNIX system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to **curses**(3X) and **terminfo**(4) in the *Programmer's Reference Manual* for more information.

Keep the manual close at hand; you'll find it invaluable when you want to know more about one of these routines or about other routines not discussed here.

Because the routines are compiled C functions, you should be familiar with the C programming language before using **curses/terminfo**. You should also be familiar with the UNIX system/C language standard I/O package (see " System Calls and Subroutines" and " Input/Output" in Chapter 2 and **stdio**(3S)). With that knowledge and an appreciation for the UNIX philosophy of building on the work of others, you can design screen management programs for many purposes.

This chapter has five sections:

- Overview

  This section briefly describes **curses**, **terminfo**, and the other components of the Terminal Information Utilities package.

- Working with **curses** Routines

  This section describes the basic routines making up the **curses**(3X) library. It covers the routines for writing to a screen, reading from a screen, and building windows. It also covers routines for more advanced screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time. Many examples are included to show the effect of using these routines.

- Working with **terminfo** Routines

  This section describes the routines in the **curses** library that deal directly with the **terminfo** database to handle certain terminal capabilities, such as programming function keys.

- Working with the **terminfo** Database

  This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

- **curses** Program Examples

  This section includes six programs that illustrate uses of **curses** routines.

Overview

What is curses?

curses(3X) is the library of routines that you use to write screen management programs on the UNIX system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw**() that behaves much like **printf**(3S) and another routine **getch**() that behaves like **getc**(3S). The automatic teller program at your bank might use **printw**() to print its menus and **getch**() to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX system screen editor **vi**(1) might also use these and other **curses** routines.

The **curses** routines are usually located in **/usr/lib/libcurses.a**. To compile a program using these routines, you must use the **cc**(1) command and include **-lcurses** on the command line so that the link editor can locate and load them:

cc *file*.**c** **-lcurses** **-o** *file*

The name **curses** comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen to update it.

For example, if you designed a screen editor program with **curses** routines and edited the sentence

```
curses/terminfo is a great package for creating screens.
```

to read

```
curses/terminfo is the best package for creating screens.
```

the program would output only `the best` in place of `a great`. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which a **curses** program is run. This means that the **curses** library can do whatever is required to update many different terminal types. It searches the **terminfo** database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your UNIX system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple **curses** program. It uses some of the basic **curses** routines to move a cursor to the middle of a terminal screen and print the character string `BullsEye`. Each of these routines is described in the following section " Working with **curses** Routines" in this chapter. For now, just look at their names and you will get an idea of what each of them does:

```
#include <curses.h>

main()
{
   initscr();

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls" );
   refresh();
   addstr("Eye" );
   refresh();
   endwin();
}
```

Figure -1. A Simple Curses Program

What Is terminfo?

terminfo refers to both of the following:

- It is a group of routines within the **curses** library that handles certain terminal capabilities. You can use these routines to program function keys, if your terminal has programmable keys, or write filters, for example. Shell programmers, as well as C programmers, can use the **terminfo** routines in their programs.

- It is a database containing the descriptions of many terminals that can be used with **curses** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

Each terminal description in the database is a separate, compiled file. You use the source code that **terminfo**(4) describes to create these files and the command **tic**(1M) to compile them.

The compiled files are normally located in the directories **/usr/lib/terminfo/?**. These directories have single character names, each of which is the first character in the name of a terminal. For example, an entry for the AT&T Teletype 5425 is normally located in the file **/usr/lib/terminfo/a/att5425**.

Here's a simple shell script that uses the **terminfo** database.

```
#   Clear the screen and show the 0,0 position.
#
tput clear
tput cup 0 0            # or tput home
echo "<- this is 0 0"

#
#   Show the 5,10 position.
#
tput cup 5 10
echo "<- this is 5 10"
```

Figure -2. A Shell Script Using terminfo Routines

How curses and terminfo Work Together

A screen management program with **curses** routines refers to the **terminfo** database at run time to obtain the information it needs about the terminal being used—what we'll call the current terminal from here on.

For example, suppose you are using an AT&T Teletype 5425 terminal to run the simple **curses** program shown in Figure 1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the `BullsEye` in the middle of it. The description of the AT&T Teletype 5425 in the database has this information. All the **curses** program needs to know before it goes looking for the information is the name of your terminal. You tell the program the name by putting it in the environment variable **$TERM** when you log in or by setting and exporting **$TERM** in your **.profile** file (see **profile**(4)). Knowing **$TERM**, a **curses** program run on the current terminal can search the **terminfo** database to find the correct terminal description.

For example, assume that the following example lines are in a **.profile**:

```
TERM=5425
export TERM
tput init
```

The first line names the terminal type, and the second line exports it. (See **profile**(4) in the *Programmer's Reference Manual.*) The third line of the example tells the UNIX system to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. (The order of these lines is important. **$TERM** must be defined and exported first, so that when **tput** is called the proper initialization for the current terminal takes place.) If you had these lines in your **.profile** and you ran a **curses** program, the program would get the information that it needs about your terminal from the file

> **/usr/lib/terminfo/a/att5425**, which provides a match for **$TERM**.

Other Components of the Terminal Information Utilities

We said earlier that the Terminal Information Utilities is commonly referred to as **curses/terminfo**. The package, however, has other components. We've mentioned some of them, for instance **tic**(1M). Here's a complete list of the components discussed in this tutorial:

captoinfo(1M)	a tool for converting terminal descriptions developed on earlier releases of the UNIX system to **terminfo** descriptions
curses(3X)	
infocmp(1M)	a tool for printing and comparing compiled terminal descriptions
tabs(1)	a tool for setting non-standard tab stops
terminfo(4)	
tic(1M)	a tool for compiling terminal descriptions for the **terminfo** database
tput(1)	a tool for initializing the tab stops on a terminal and for outputting the value of a terminal capability

We also refer to **profile**(4), **scr_dump**(4), **term**(4), and **term**(5). For more information about any of these components, see the *Programmer's Reference Manual* and the *User's Reference Manual.*

Working with curses Routines

This section describes the basic routines for creating interactive screen management programs. It begins by describing the routines and other program components that every program needs to work properly. Then it tells you how to compile and run a program. Finally, it describes the most frequently used routines that

- write output to and read input from a terminal screen
- control the data output and input — for example, to print output in bold type or prevent it from echoing (printing back on a screen)
- manipulate multiple screen images (windows)
- draw simple graphics
- manipulate soft labels on a terminal screen
- send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, we include simple example programs as the routines are introduced. We also refer to a group of larger examples located in the section " **curses** Program Examples" in this chapter. These larger examples are more challenging; they sometimes make use of routines not discussed here. Keep the **curses**(3X) manual page handy.

What Every curses Program Needs

All programs need to include the header file <**curses.h**> and call the routines **initscr**(), **refresh**() or similar related routines, and **endwin**().

The Header File <curses.h>

The header file <**curses.h**> defines several global variables and data structures and defines several routines as macros.

To begin, let's consider the variables and data structures defined. <**curses.h**> defines all the parameters used by routines. It also defines the integer variables **LINES** and **COLS**; when a program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine **initscr**() described below. The header file defines the constants **OK** and **ERR**, too. Most routines have return values; the **OK** value is returned if a routine is properly completed, and the **ERR** value if some error occurs.

> *Note:* **LINES** and **COLS** are external (global) variables that represent the size of a terminal screen. Two similar variables, **$LINES** and **$COLUMNS**, may be set in a user's shell environment; a **curses** program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this chapter, we will use the **$** to distinguish them from the C declarations in the <**curses.h**> header file.

For more information about these variables, see the following sections " The Routines **initscr(), refresh**(), and **endwin**()" and " More about **initscr**() and Lines and Columns."

Now let's consider the macro definitions. <**curses.h**> defines many **curses** routines as macros that call other macros or **curses** routines. For instance, the simple routine **refresh**() is a macro.

The line

```
#define refresh() wrefresh(stdscr)
```

shows when **refresh** is called, it is expanded to call the routine **wrefresh**(). The latter routine in turn calls the two routines **wnoutrefresh**() and **doupdate**(). Many other routines also group two or three routines together to achieve a particular result.

> *Caution: Macro expansion in* **curses** programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about <**curses.h**>: it automatically includes <**stdio.h**> and the <**termio.h**> tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines initscr(), refresh(), endwin()

The routines **initscr**(), **refresh**(), and **endwin**() initialize a terminal screen to an " in state," update the contents of the screen, and restore the terminal to an " out of state," respectively. Use the simple program that we introduced earlier to learn about each of these routines.

```
#include <curses.h>

main()
{
   initscr();    /* initialize terminal settings and <curses.h>
                    data structures and variables */

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();    /* send output to (update) terminal screen */
   addstr("Eye");
   refresh();    /* send more output to terminal screen */
   endwin();     /* restore all terminal settings */
}
```

Figure -3. The Purposes of initscr(), refresh(), and **endwin()** in a Program

A **curses** program usually starts by calling **initscr()**; the program should call **initscr()** only once. Using the environment variable **$TERM** as the section " How **curses** and **terminfo** Work Together" describes, this routine determines what terminal is being used. It then initializes all the declared data structures and other variables from **<curses.h>**. For example, **initscr()** would initialize **LINES** and **COLS** for the sample program on whatever terminal it was run. If the Teletype 5425 were used, this routine would initialize **LINES** to 24 and **COLS** to 80. Finally, this routine writes error messages to **stderr** and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move()** and **addstr()** in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. Then the line

```
addstr("Bulls");
```

says to write the character string `Bulls`. For example, if the Teletype 5425 were used, these routines would position the cursor and write the character string at (11,36).

> *Note:* All **curses** routines that move the cursor move it from its home position in the upper left corner of a screen. The **(LINES,COLS)** coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common 'x,y' order of screen (or graph) coordinates. The -1 in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like **move**() and **addstr**() do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh**() is called. Before this, an internal representation of the screen called a window is updated.

This is a very important concept, which we discuss below under " More about **refresh**() and Windows."

Finally, a program ends by calling **endwin**(). This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling a curses Program

You compile programs that include **curses** routines as C language programs using the **cc**(1) command (documented in the *Programmer's Reference Manual*), which invokes the C compiler (see Chapter 2 in this guide for details).

The routines are usually stored in the library **/usr/lib/libcurses.a**. To direct the link editor to search this library, you must use the **–l** option with the **cc** command.

The general command line for compiling a **curses** program follows:

cc *file*.**c** **–lcurses** **–o** *file*

file.**c** is the name of the source program; and *file* is the executable object module.

Running a curses Program

curses programs count on certain information being in a user's environment to run properly. Specifically, users of a program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
tput init
```

For an explanation of these lines, see the section " How **curses** and **terminfo** Work Together" in this chapter. Users of a **curses** program could also define the environment variables **$LINES**, **$COLUMNS**, and **$TERMINFO** in their **.profile** files. However, unlike **$TERM**, these variables do not have to be defined.

If a **curses** program does not run as expected, you might want to debug it with **sdb**(1), which is documented in the *Programmer's Reference Manual*). When using **sdb**, you have to keep a few points in mind. First, a **curses** program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **sdb**, however, may cause changes to the contents of the screen of which the **curses** program is not aware.

Second, a **curses** program outputs to a window until **refresh**() or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on routines that are macros, such as **refresh**(), does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh**() instead of **refresh**(). See the above section, " The Header File <**curses.h**>," for more information about macros.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, **initscr**() sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment **$LINES** and **$COLUMNS**.

More about refresh() and Windows

As mentioned above, routines do not update a terminal until **refresh**() is called. Instead, they write to an internal representation of the screen called a window. When **refresh**() is called, all the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like a buffer does when you use a UNIX system editor. When you invoke **vi**(1), for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of **curses** routines, they change the contents of a window. The changes become part of the current terminal screen only when **refresh**() is called.

<**curses.h**> supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using routines. The header file defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh**() is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like **stdscr**. A **curses** program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as is possible. Figure 4 illustrates what happens when you execute the sample curses program that prints `BullsEye` at the center of a terminal screen (see Figure 1). Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh**() is called. This **refresh**() clears the screen and updates it with the current contents of **stdscr**.

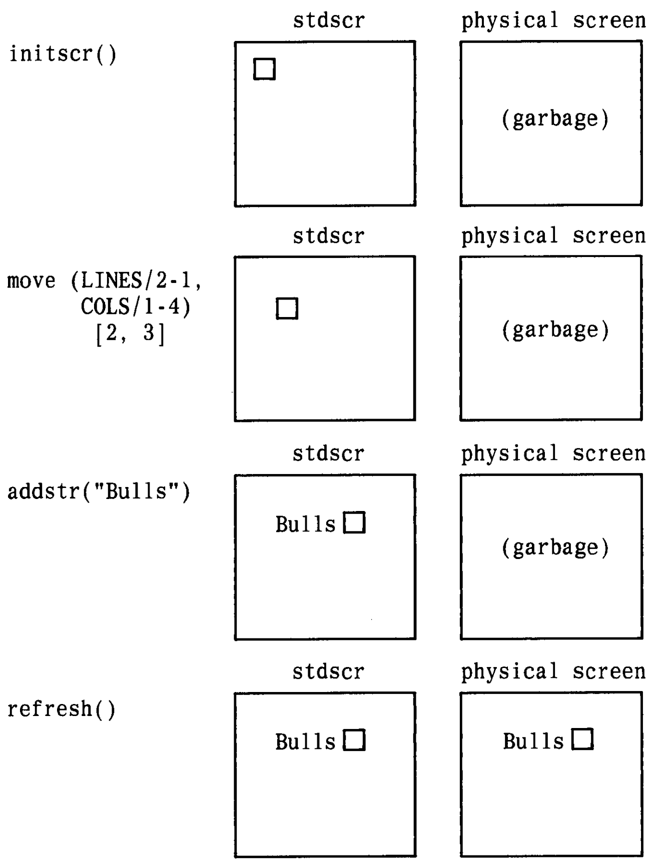


Figure -4. The Relationship between *stdscr* and a Terminal Screen (Sheet 1 of 2)

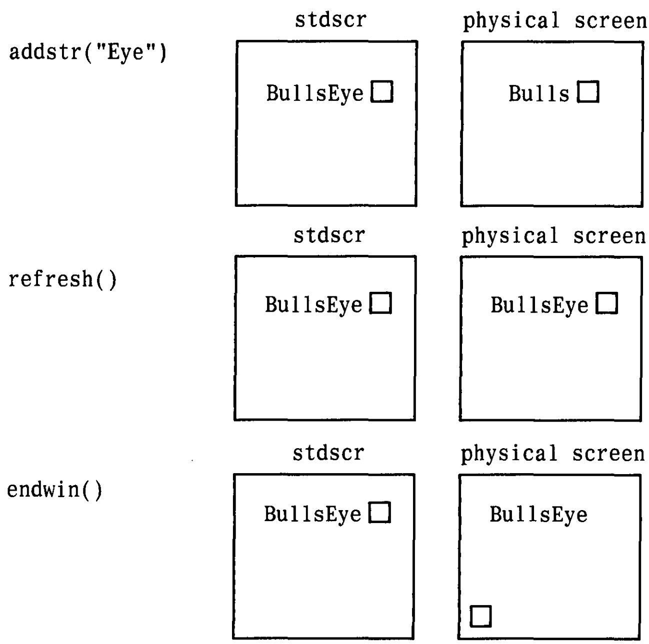


Figure -4. The Relationship Between *stdscr* and a Terminal Screen (Sheet 2 of 2)

You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It's possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It's also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some **curses** routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 5 represents what a pad, a subwindow, and some other windows could look like in comparison to a terminal screen.

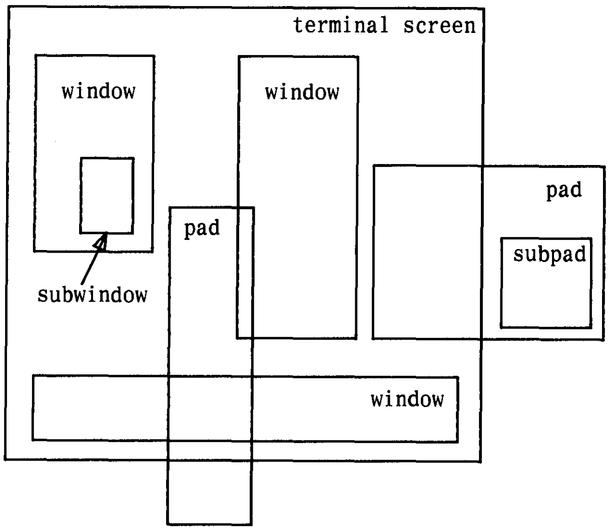


Figure -5. Multiple Windows and Pads Mapped to a Terminal Screen

The section " Building Windows and Pads" in this chapter describes the routines you use to create and use them. If you'd like to see a **curses** program with windows now, you can turn to the **window** program under the section " **curses** Program Examples" in this chapter.

Getting Simple Output and Input

Output

The routines that provides for writing to **stdscr** are similar to those provided by the **stdio**(3S) library for writing to a file. They let you

- write a character at a time — **addch()**
- write a string — **addstr()**
- format a string from a variety of input arguments — **printw()**
- move a cursor or move a cursor and print character(s) — **move()**, **mvaddch()**, **mvaddstr()**, **mvprintw()**
- clear a screen or a part of it — **clear()**, **erase()**, **clrtoeol()**, **clrtobot()**

Following are descriptions and examples of these routines.

> *Caution: The* **curses** library provides its own set of output and input functions. You should not use other I/O routines or system calls, like **read**(2) and **write**(2), in a **curses** program. They may cause undesirable results when you run the program.

addch()

SYNOPSIS

#include <curses.h>

int addch(ch)
chtype ch;

NOTES

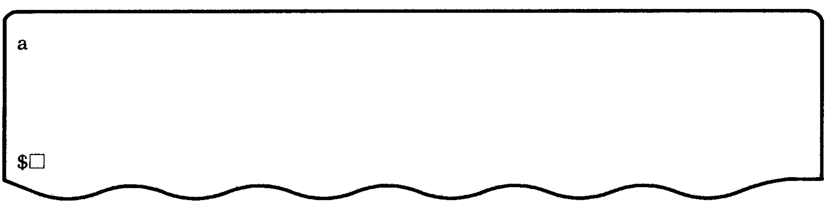
- **addch()** writes a single character to **stdscr**.
- The character is of the type **chtype**, which is defined in <**curses.h**>. **chtype** contains data and attributes (see " Output Attributes" in this chapter for information about attributes).
- When working with variables of this type, make sure you declare them as **chtype** and not as the basic type (for example, **short**) that **chtype** is declared to be in <**curses.h**>. This will ensure future compatibility.
- **addch()** does some translations. For example, it converts
  1. the <**NL**> character to a clear to end of line and a move to the next line
  2. the tab character to an appropriate number of blanks
  3. other control characters to their ^X notation
- **addch()** normally returns **OK**. The only time **addch()** returns **ERR** is after adding a character to the lower right-hand corner of a window that does not scroll.
- **addch()** is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addch('a');
    refresh();
    endwin();
}
```

The output from this program will appear as follows:



Also see the **show** program under " **curses** Example Programs" in this chapter.

addstr()

SYNOPSIS

```
#include <curses.h>

int addstr(str)
char *str;
```

NOTES

- **addstr()** writes a string of characters to **stdscr**.
- **addstr()** calls **addch()** to write each character.
- **addstr()** follows the same translation rules as **addch()**.
- **addstr()** returns **OK** on success and **ERR** on error.
- **addstr()** is a macro.

EXAMPLE

Recall the sample program that prints the character string `BullsEye`. See Figures 1, 2, and 4.

printw()

SYNOPSIS

#include <curses.h>

int printw(fmt [,arg...])
char *fmt

NOTES

- **printw()** handles formatted printing on **stdscr**.
- Like **printf**, **printw()** takes a format string and a variable number of arguments.
- Like **addstr()**, **printw()** calls **addch()** to write the string.
- **printw()** returns **OK** on success and **ERR** on error.

EXAMPLE

```
#include <curses.h>

main()
{
   char* title = "Not specified";
   int no = 0;

      /* Missing code. */

   initscr();

      /* Missing code. */

   printw("%s is not in stock.\n", title);
   printw("Please ask the cashier to order %d for you.\n",
      no);

   refresh();
   endwin();
}
```

The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.


$□
```

move()

SYNOPSIS

#include <curses.h>

int move(y, x);
int y, x;

NOTES

- **move**() positions the cursor for **stdscr** at the given row **y** and the given column **x**.

- Notice that **move**() takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section " The Routines **initscr**(), **refresh**(), and **endwin**()" for more information.

- **move**() may be combined with the write functions to form

  1. **mvaddch**(**y, x, ch**), which moves to a given position and prints a character

  2. **mvaddstr**(**y, x, str**), which moves to a given position and prints a string of characters

  3. **mvprintw**(**y, x, fmt** [,*arg*...]), which moves to a given position and prints a formatted string.

- **move**() returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (**LINES** - 1, **COLS** - 1) causes an error.

- **move**() is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();      /* Gets <CR>; discussed below. */
    endwin();
}
```

Here's the output generated by running this program:

```
Cursor should be here -->□if move() works.


Press <CR> to end test.
```

After you press <**CR**>, the screen looks like this:

```
Cursor should be here -->


Press <CR> to end test.
$□
```

See the **scatter** program under " **curses** Program Examples" in this chapter for another example of using **move**().

clear() and erase()

SYNOPSIS

#include <curses.h>

int clear()
int erase()

NOTES

- Both routines change **stdscr** to all blanks.
- **clear()** also assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase()** and then **clearok()** which clears the physical screen completely on the next call to **refresh()** for **stdscr**. See the **curses**(3X) manual page for more information about **clearok()**.
- **initscr()** automatically calls **clear()**.
- **clear()** always returns **OK**; **erase()** returns no useful value.
- Both routines are macros.

clrtoeol() and clrtobot()

SYNOPSIS

#include <curses.h>

int clrtoeol()
int clrtobot()

NOTES

- **clrtoeol**() changes the remainder of a line to all blanks.
- **clrtobot**() changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

EXAMPLE

The following sample program uses **clrtobot()**.

```
#include <curses.h>

main()
{
   initscr();
   addstr("Press <CR> to delete from here to the end \
of the line and on.");
   addstr("\nDelete this too.\nAnd this.");
   move(0,30);
   refresh();
   getch();
   clrtobot();
   refresh();
   endwin();
}
```

Here's the output generated by running this program:

```
Press <CR> to delete from here□to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to **refresh()**: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press <CR>:

```
Press <CR> to delete from here

$□
```

See the **show** and **two** programs under " **curses** Example Programs" for examples of uses for **clrtoeol**().

Input

routines for reading from the current terminal are similar to those provided by the **stdio**(3S) library for reading from a file. They let you

- read a character at a time — **getch**()
- read a <**NL**>-terminated string — **getstr**()
- parse input, converting and assigning selected data to an argument list — **scanw**()

The primary routine is **getch**(), which processes a single input character and then returns that character. This routine is like the C library routine **getchar**()(3S) except that it makes several terminal- or system-dependent options available that are not possible with **getchar**(). For example, you can use **getch**() with the **curses** routine **keypad**(), which allows a program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch**() and **keypad**() on the **curses**(3X) manual page for more information about **keypad**().

getch() SYNOPSIS

#include <curses.h>

int getch()

NOTES

- **getch**() reads a single character from the current terminal.
- **getch**() returns the value of the character or **ERR** on 'end of file,' receipt of signals, or non-blocking read with no input.
- **getch**() is a macro.
- See the discussions about **echo(), noecho(), cbreak(), nocbreak(), raw(), noraw(), halfdelay(), nodelay(),** and **keypad**() below and in **curses**(3X).

The following pages describe and give examples of the basic routines for getting input in a screen program.

EXAMPLE

```
#include <curses.h>

main()
{
    int ch;

    initscr();
    cbreak();
    addstr("Press any character:  ");
    refresh();
    ch = getch();
    printw("\n\n\nThe character entered was a '%c'.\n",
        ch);
    refresh();
    endwin();
}
```

The output from this program follows. The first **refresh()** sends the **addstr()** character string from **stdscr** to the terminal:

```
Press any character:  □
```

Then assume that a **w** is typed at the keyboard. **getch()** accepts the character and assigns it to **ch**. Finally, the second **refresh()** is called and the screen appears as follows:

```
Press any character:  w


The character entered was a 'w'.


$□
```

For another example of **getch()**, see the **show** program under " **curses** Example Programs" in this chapter.

getstr()

SYNOPSIS

#include <curses.h>

int getstr(str)
char *str;

NOTES

- **getstr**() reads characters and stores them in a buffer until a <CR>, <NL>, or <ENTER> is received from **stdscr**. **getstr**() does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr**() is a macro; it calls **getch**() to read each character.
- **getstr**() returns **ERR** if **getch**() returns **ERR** to it. Otherwise it returns **OK**.
- See the discussions about **echo**(), **noecho**(), **cbreak**(), **nocbreak**(), **raw**(), **noraw**(), **halfdelay**(), **nodelay**(), and **keypad**() below and in **curses**(3X).

EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

  initscr();
  cbreak();
  addstr("Enter a character string terminated by <CR>:\n\n");
  refresh();
  getstr(str);
  printw("\n\n\nThe string entered was \n'%s'\n", str);
  refresh();
  endwin();
}
```

Assume you entered the string 'I enjoy learning about the UNIX system.' The final screen (after entering **<CR>**) would appear as follows:

```
Enter a character string terminated by <CR>:

I enjoy learning about the UNIX system.


The string entered was
'I enjoy learning about the UNIX system.'

$□
```

scanw()

SYNOPSIS

#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;

NOTES

- **scanw**() calls **getstr**() and parses an input line.
- Like **scanf**(3S), **scanw**() uses a format string to convert and assign to a variable number of arguments.
- **scanw**() returns the same values as **scanf**().
- See **scanf**(3S) for more information.

EXAMPLE

```
#include <curses.h>

main()
{
   char string[100];
   float number;

   initscr();
   cbreak();            /* Explained later in the   */
   echo();              /* section "Input Options"  */
   addstr("Enter a number and a string separated by \
a comma: ");
   refresh();
   scanw("%f,%s",&number,string);
   clear();
   printw("The string was \"%s\" and the number was %f.",
           string,number);
   refresh();
   endwin();
}
```

Notice the two calls to **refresh()**. The first call updates the screen with the character string passed to **addstr()**, the second with the string returned from **scanw()**. Also notice the call to **clear()**. Assume you entered the following when prompted: **2,twin**. After running this program, your terminal screen would appear, as follows:

```
The string was "twin" and the number was 2.000000.

$□
```

Controlling Output and Input

Output Attributes

When we talked about **addch**(), we said that it writes a single character of the type **chtype** to **stdscr**. **chtype** has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, bold, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine **attrset**() and related **curses** routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- A_BLINK — blinking
- A_BOLD — extra bright or bold
- A_DIM — half bright
- A_REVERSE — reverse video
- A_STANDOUT — a terminal's best highlighting mode
- A_UNDERLINE — underlining
- A_ALTCHARSET — alternate character set (see the section " Drawing Lines and Other Graphics" in this chapter)

To use these attributes, you must pass them as arguments to **attrset**() and related routines; they can also be ORed with the bitwise OR (|) to **addch**().

Note: Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a **curses** program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:

```
 ...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
 ...
refresh();
```

Attributes can be turned on singly, such as **attrset**(A_BOLD) in the example, or in combination. To turn on blinking bold text, for example, you would use **attrset**(A_BLINK ¦ A_BOLD). Individual attributes can be turned on and off with the **curses** routines **attron**() and **attroff**() without affecting other attributes. **attrset(0)** turns all attributes off.

Notice the attribute called A_STANDOUT. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, **standout**() and **standend**() can be used to turn on and off this attribute. **standend**(), in fact, turns of all attributes.

In addition to the attributes listed above, there are two bit masks called A_CHARTEXT and A_ATTRIBUTES. You can use these bit masks with the **curses** function **inch**() and the C logical AND (**&**) operator to extract the character or attributes of a position on a terminal screen. See the discussion of **inch**() on the **curses**(3X) manual page.

Following are descriptions of **attrset**() and the other **curses** routines that you can use to manipulate attributes.

attron(), attrset(), and attroff()

SYNOPSIS

#include <curses.h>

int attron(attrs)
chtype attrs;

int attrset(attrs)
chtype attrs;

int attroff(attrs)
chtype attrs;

NOTES

- **attron**() turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in <**curses.h**>.

- **attrset**() turns on the requested attributes **attrs** instead of any that are currently turned on.

- **attroff**() turns off the requested attributes **attrs** if they are on.

- The attributes may be combined using the bitwise OR (|).

- All return **OK**.

EXAMPLE

See the **highlight** program under " **curses** Example Programs" in this chapter.

standout() and standend()

SYNOPSIS

#include <curses.h>

int standout()
int standend()

NOTES

- **standout**() turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to **attron**(A_STANDOUT).

- **standend**() turns off all attributes. This routine is equivalent to **attrset**(0).

- Both always return **OK**.

EXAMPLE

See the **highlight** program under " **curses** Example Programs" in this chapter.

Bells, Whistles, and Flashing Lights

Occasionally, you may want to get a user's attention. Two routines were designed to help you do this. They let you ring the terminal's chimes and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine **beep**() can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep**() will flash the screen.)

beep() and flash()

SYNOPSIS

#include <curses.h>

int flash()
int beep()

NOTES

- **flash**() tries to flash the terminals screen, if possible, and, if not, tries to ring the terminal bell.
- **beep**() tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- Neither returns any useful value.

Input Options

The UNIX system does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character (typically #) and a line kill character (typically @)
- interprets a CTRL-D (control d) as end of file (EOF)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates <**CR**> to <**NL**>

Because a **curses** program maintains total control over the screen, **curses** turns off echoing on the UNIX system and does echoing itself. At times, you may not want the UNIX system to process other characters in the standard way in an interactive screen management program. Some **curses** routines, **noecho**() and **cbreak**(), for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Figure 6 shows some of the major routines for controlling input.

Every program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak**(), **raw**(), **nocbreak**(), or **noraw**() mode. Although the program starts up in **echo**() mode, as Figure 6 shows, none of the other modes are guaranteed.

The combination of **noecho**() and **cbreak**() is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The **curses** routine **noecho**() is designed for this purpose. However, when **noecho**() turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak**() causes these characters to be uninterpreted.

Input Options	Characters Interpreted	Characters Uninterpreted
Normal 'out of **curses** state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal **curses** 'start up state'	echoing (simulated)	All else undefined.
cbreak() and **echo**()	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and **noecho**()	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and **noecho**()	break, quit stripping erase, kill EOF	echoing
nocbreak() and **echo**()	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of **cbreak**())		break, quit stripping

Figure -6. Input Option Settings for curses Programs

> *Caution: Do not use the combination* **nocbreak**() and **noecho**(). If you use it in a program and also use **getch**(), the program will go in and out of **cbreak**() mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Figure 6, you can use the **curses** routines **noraw(), halfdelay**(), and **nodelay**() to control input. See the **curses**(3X) manual page for discussions of these routines.

The next few pages describe **noecho**(), **cbreak**() and the related routines **echo**() and **nocbreak**() in more detail.

echo() and noecho()

SYNOPSIS

#include <curses.h>

int echo()
int noecho()

NOTES

- **echo**() turns on echoing of characters by **curses** as they are read in. This is the initial setting.

- **noecho**() turns off the echoing.

- Neither returns any useful value.

- **curses** programs may not run properly if you turn on echoing with **nocbreak**(). See Figure 6 and accompanying caution. After you turn echoing off, you can still echo characters with **addch**().

EXAMPLE

See the **editor** and **show** programs under " **curses** Program Examples" in this chapter.

cbreak() and nocbreak()

SYNOPSIS

#include < curses.h >
int cbreak()
int nocbreak()

NOTES

- **cbreak**() turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- **nocbreak**() returns to normal 'line at a time' processing. This is typically the initial setting.
- Neither returns any useful value.
- A **curses** program may not run properly if **cbreak**() is turned on and off within the same program or if the combination **nocbreak**() and **echo**() is used.
- See Figure 6 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under " **curses** Program Examples" in this chapter.

Building Windows and Pads

An earlier section in this chapter, " More about **refresh**() and Windows" explained what windows and pads are and why you might want to use them. This section describes the **curses** routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch**('c') would become **waddch(mywin, 'c')** if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in " Getting Simple Output and Input."

- **waddch(***win, ch***)**
- **mvwaddch(***win, y, x, ch***)**
- **waddstr(***win, str***)**
- **mvwaddstr(***win, y, x, str***)**
- **wprintw(***win, fmt [, arg...]***)**
- **mvwprintw(***win, y, x, fmt [, arg...]***)**
- **wmove(***win, y, x***)**

- **wclear(*win*)** and **werase(*win*)**
- **wclrtoeol(*win*)** and **wclrtobot(*win*)**
- **wrefresh**()

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the addition of a *win* argument. Notice that the routines whose names begin with **mvw** take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See **curses**(3X) for more information about these routines or the versions of the input routines **getch, getstr**(), and so on that you should use with windows.

All **w** routines can be used with pads except for **wrefresh**() and **wnoutrefresh**() (see below). In place of these two routines, you have to use **prefresh**() and **pnoutrefresh**() with pads.

The Routines wnoutrefresh() and doupdate()

If you recall from the earlier discussion about **refresh**(), we said that it sends the output from **stdscr** to the terminal screen. We also said that it was a macro that expands to **wrefresh(stdscr)** (see " What Every **curses** Program Needs" and " More about **refresh**() and Windows").

The **wrefresh**() routine is used to send the contents of a window (**stdscr** or one that you create) to a screen; it calls the routines **wnoutrefresh**() and **doupdate**(). Similarly, **prefresh**() sends the contents of a pad to a screen by calling **pnoutrefresh**() and **doupdate**().

Using **wnoutrefresh()**—or **pnoutrefresh()** (this discussion will be limited to the former routine for simplicity)—and **doupdate()**, you can update terminal screens with more efficiency than using **wrefresh()** by itself. **wrefresh()** works by first calling **wnoutrefresh()**, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh()**, **wrefresh()** then calls **doupdate()**, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to a screen. However, by calling **wnoutrefresh()** for each window and then **doupdate()** only once, you can minimize the total number of characters transmitted and the processor time used. The following sample program uses only one **doupdate()**:

```
#include <curses.h>

main()
{
   WINDOW *w1, *w2;

   initscr();
   w1 = newwin(2,6,0,3);
   w2 = newwin(1,4,5,4);
   waddstr(w1, "Bulls");
   wnoutrefresh(w1);
   waddstr(w2, "Eye");
   wnoutrefresh(w2);
   doupdate();
   endwin();
}
```

Notice from the sample that you declare a new window at the beginning of a **curses** program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named w1 and w2 with the routine **newwin()** according to certain specifications. **newwin()** is discussed in more detail below.

Figure 7 illustrates the effect of **wnoutrefresh()** and **doupdate()** on these two windows, the virtual screen, and the physical screen:

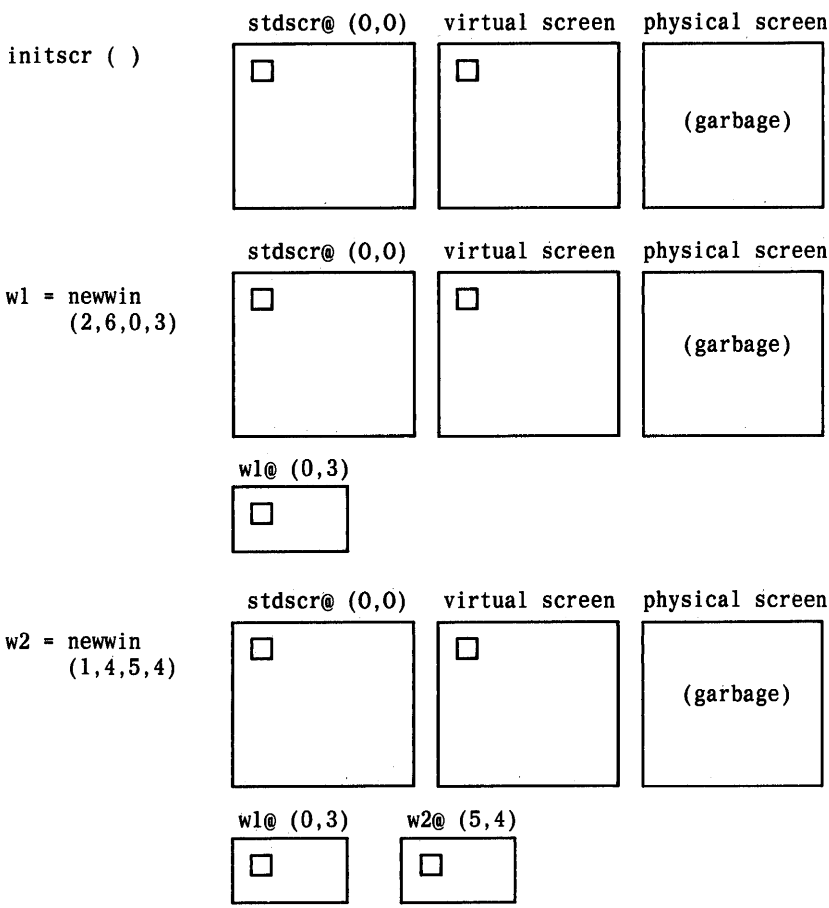


Figure -7. The Relationship Between a Window and a Terminal Screen (Sheet 1 of 3)

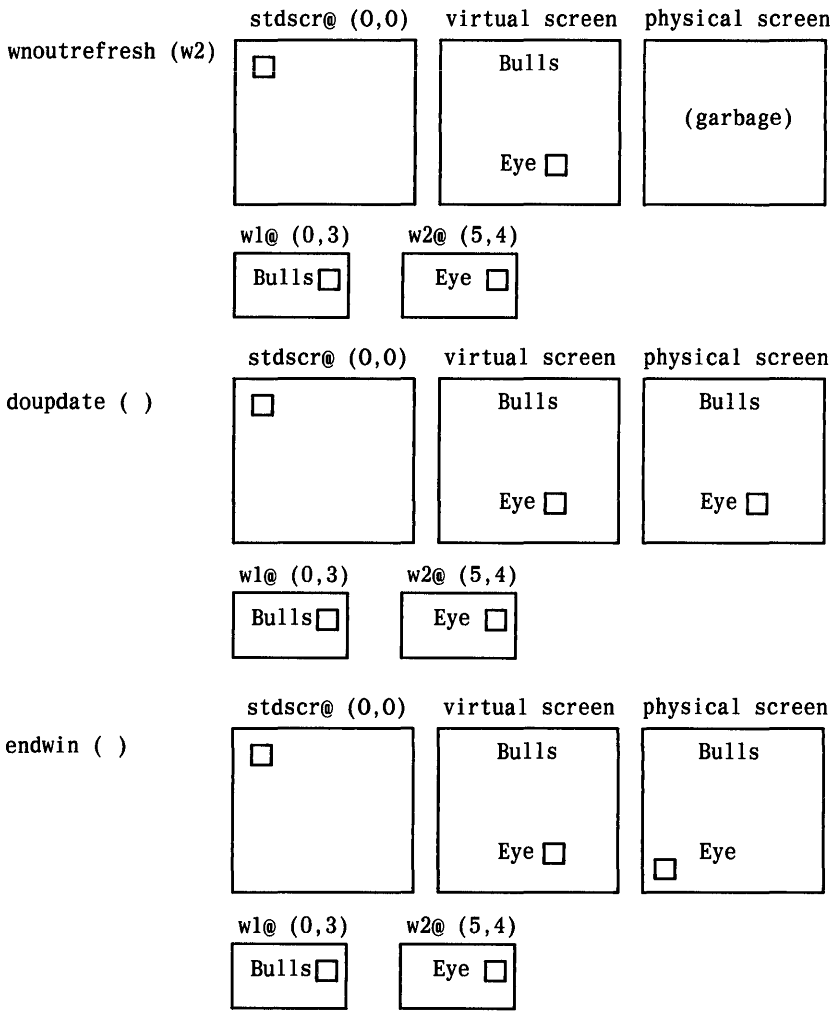


Figure -7. The Relationship Between a Window and a Terminal Screen (Sheet 2 of 3)

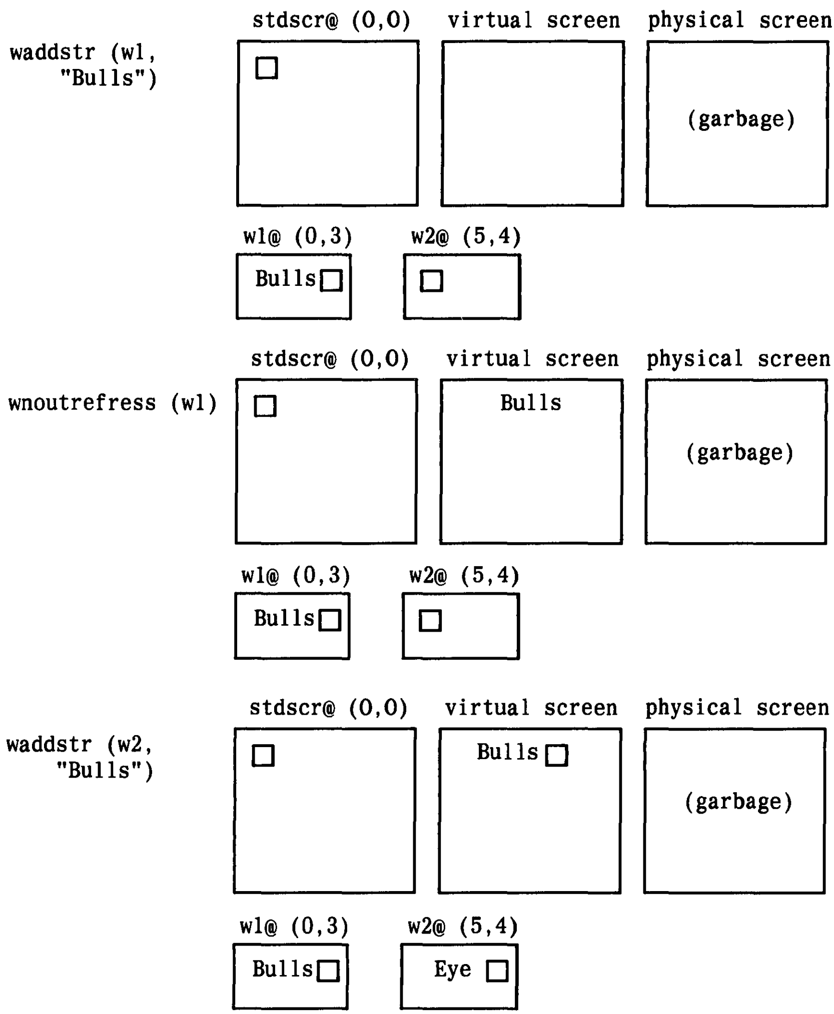


Figure -7. The Relationship Between a Window and a Terminal Screen (Sheet 3 of 3)

New Windows

Following are descriptions of the routines **newwin**() and **subwin**(), which you use to create new windows. For information about creating new pads with **newpad**() and **subpad**(), see the **curses**(3X) manual page.

newwin()

SYNOPSIS

#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;

NOTES

- **newwin()** returns a pointer to a new window with a new data area.
- The variables **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

EXAMPLE

Recall the sample program using two windows; see Figure 7. Also see the **window** program under " **curses** Program Examples" in this chapter.

subwin()

SYNOPSIS

```
#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

NOTES

- **subwin**() returns a new window that points to a section of another window, **orig**.
- **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.

Caution: Subwindows of subwindows do not work (as of the copyright date of this Programmer's Guide).

EXAMPLE

```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w');
  mvwaddstr(stdscr,7,10,"------- this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of **w**s around the stdscr (the sides of your terminal screen) and a border of s's around the subwindow **sub** when it is run. For another example, see the **window** program under " **curses** Program Examples" in this chapter.

Using Advanced curses Features

Knowing how to use the basic **curses** routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The **curses** library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single **curses** program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the **curses**(3X) manual page before you try to use the advanced **curses** features.

> ***Caution: The routines described under " Routines for Drawing Lines and Other Graphics" and " Routines for Using Soft Labels" are features that are new for UNIX System V Release 3.0. If a program uses any of these routines, it may not run on earlier releases of the UNIX system. You must use the Release 3.0 version of the library on UNIX System V Release 3.0 to work with these routines.***

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in **curses** programs. **curses** use the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in a **curses** program, you pass a set of variables whose names begin with ACS_ to the **curses** routine **waddch**() or a related routine. For example, ACS_ULCORNER is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, ACS_ULCORNER's value is the terminal's character for that glyph OR'd (|) with the bit-mask A_ALTCHARSET. If no line drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_ names and their defaults are listed on the **curses**(3X) manual page.

Part of an example program that uses line drawing characters follows. The example uses the **curses** routine **box**() to draw a box around a menu on a screen. **box**() uses the line drawing characters by default or when | (the pipe) and – are chosen. (See **curses**(3X).) Up and down more indicators are drawn on the box border (using **ACS_UARROW** and **ACS_DARROW**) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);
 ...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

For more information, see **curses**(3X) in the *Programmer's Reference Manual.*

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The **curses** library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a **curses** program to make use of them.

Let's briefly discuss most of the **curses** routines needed to use soft labels: **slk_init**(), **slk_set**(), **slk_refresh**() and **slk_noutrefresh**(), **slk_clear**, and **slk_restore**.

When you use soft labels in a **curses** program, you have to call the routine **slk_int**() before **initscr**(). This sets an internal flag for **initscr**() to look at that says to use the soft labels. If **initscr**()

discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels. The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The **curses** routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk_set**() takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk_noutrefresh**() is comparable to **wnoutrefresh**() in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh**() commonly follows, **slk_noutrefresh**() is the function that is most commonly used to output the labels.

Just as **wrefresh**() is equivalent to a **wnoutrefresh**() followed by a **doupdate**(), so too the function **slk_refresh**() is equivalent to a **slk_noutrefresh**() followed by a **doupdate**().

To prevent the soft labels from getting in the way of a shell escape, **slk_clear**() may be called before doing the **endwin**(). This clears the soft labels off the screen and does a **doupdate**(). The function **slk_restore**() may be used to restore them to the screen.

See the **curses**(3X) manual page for more information about the routines for using soft labels.

Working with More than One Terminal

A **curses** program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the **curses** library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking **$TERM** in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

A **curses** program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals in a **curses** program have the type **SCREEN***. A new terminal is initialized by calling **newterm(***type, outfd, infd***)**. **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio**(3S) file pointer (**FILE***) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr**(), which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call **set_term(***sp)* where *sp* is the screen reference to be made current. **set_term**() returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**(). Options such as **cbreak**() and **noecho**() must be set separately for each terminal. The functions **endwin**() and **refresh**() must be called separately for each terminal. Figure 8 shows a typical scenario to output a message to several terminals.

```
for (i=0; i<nterm; i++)
{
   set_term(terms[i]);
   mvaddstr(0, 0, "Important message");
   refresh();
}
```

Figure -8. Sending a Message to Several Terminals

See the **two** program under " **curses** Program Examples" in this chapter for a more complete example.

Working with terminfo Routines

Some programs need to use lower level routines (i.e., primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines:

the higher level **curses** routines make your program more portable to other UNIX systems and to a wider class of terminals.

> *Note:* You are discouraged from using **terminfo** routines except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Figure 9.

```
#include <curses.h>
#include <term.h>
...
   setupterm( (char*)0, 1, (int*)0 );
   ...
   putp(clear_screen);
   ...
   reset_shell_mode();
   exit(0);
```

Figure -9. Typical Framework of a terminfo Program

The header files <**curses.h**> and <**term.h**> are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm()** takes care of initialization. Passing this routine the values **(char*)0, 1,** and **(int*)0** invokes reasonable defaults. If **setupterm()** can't figure out what kind of terminal you are on, it prints an error message and exits. **reset_shell_mode**() performs functions similar to **endwin**() and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm**(). It can be output using the **terminfo** routines **putp**() or **tputs**(), which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf**(3S), because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch**() and **getch**() are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4); see **curses**(3X) for a list of all the **terminfo** routines.

Compiling and Running a terminfo Program

The general command line for compiling and the guidelines for running a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections " Compiling a **curses** Program" and " Running a **curses** Program" in this chapter for more information.

An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see " **curses** Program Examples") that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;  /* Currently underlining */

main(argc, argv)
 int argc;
 char **argv;
{
  FILE *fd;
  int c, c2;
  int outch();

  if (argc > 2)
  {
    fprintf(stderr, "Usage: termhl [file]\n");
    exit(1);
  }

  if (argc == 2)
  {
    fd = fopen(argv[1], "r");
    if (fd == NULL)
    {
      perror(argv[1]);
      exit(2);
    }
  }
  else
  {
    fd = stdin;
  }
  setupterm((char*)0, 1, (int*)0);
```

```
	for (;;)
	{
		c = getc(fd);
		if (c == EOF)
		break;
		if (c == '\')
		{
			c2 = getc(fd);
			switch (c2)
			{
				case 'B':
				tputs(enter_bold_mode, 1, outch);
				continue;
				case 'U':
				tputs(enter_underline_mode, 1, outch);
				ulmode = 1;
				continue;
				case 'N':
				tputs(exit_attribute_mode, 1, outch);
				ulmode = 0;
				continue;
			}
			putch(c);
			putch(c2);
		}
		else
			putch(c);
	}
	fclose(fd);
	fflush(stdout);
	resetterm();
	exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
```

```
  int c;
{
  outch(c);
  if (ulmode && underline_char)
  {
    outch('\b');
    tputs(underline_char, 1, outch);
  }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
  int c;
{
  putchar(c);
}
```

Let's discuss the use of the function **tputs(***cap, affcnt, outc***)** in this program to gain some insight into the **terminfo** routines. **tputs**() applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the **terminfo** database probably contain strings like **$<20>**, which means to pad for 20 milliseconds (see the following section " Specify Capabilities" in this chapter). **tputs** generates enough pad characters to delay for the appropriate time.

tput() has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls **putchar**. For these programs, the routine **putp(***cap***)** is a convenient abbreviation. **termhl** could be simplified by using **putp()**.

Now to understand why you should use the **curses** level routines instead of **terminfo** level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low level details such as this are precisely why the **curses** level is recommended over the terminfo level. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** (see **curses**(3X)) could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some UNIX system tools, like **vi**(1), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already described in the **terminfo** database. However, it is possible that you'll want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier. (Lest we forget the UNIX motto: Build on the work of others.)

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named " myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (|). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a **-w**) version of our fictitious terminal would be described as **myterm-w**. **term**(5) describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.

- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways — type:

  stty -echo; cat -vu
  Type in the keys you want to test;
  for example, see what right arrow (→) transmits.
  <CR>
  <CTRL-D>

stty echo

or

cat >/dev/null
Type in the escape sequences you want to test;
for example, see what **\E [H** transmits.
<CTRL-D>

- The first line in each of these testing methods sets up the terminal to carry out the tests. The **<CTRL-D>** helps return the terminal to its normal settings.

- See the **terminfo**(4) manual page. It lists all the capability names you have to use in a terminal description. The following section, " Specify Capabilities," gives details.

Specify Capabilities

Once you know the capabilities of your terminal, you have to describe them in your terminal description. You describe them with a string of comma-separated fields that contain the abbreviated **terminfo** name and, in some cases, the terminal's value for each capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as **bel=^G,**.

The list of capabilities may continue onto multiple lines as long as white space (that is, tabs and spaces) begins every line but the first of the description. Comments can be included in the description by putting a **#** at the beginning of the line.

The **terminfo**(4) manual page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled " Capname."

> *Note:* For a **curses** program to run on any given terminal, its description in the **terminfo** database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

#	This shows a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as **cols#80,**.
=	This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:
^	This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as **^G**.

\E or \e These characters followed by another character show an escape instruction. An entry of \EC would transmit to the terminal as ESCAPE-C.

\n These characters provide a <**NL**> character sequence.

\l These characters provide a linefeed character sequence.

\r These characters provide a return character sequence.

\t These characters provide a tab character sequence.

\b These characters provide a backspace character sequence.

\f These characters provide a formfeed character sequence.

\s These characters provide a space character sequence.

nnn This is a character whose three-digit octal is *nnn,* where *nnn* can be one to three digits.

$< > These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the " less than/greater than" symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as **$<20*>**. See the **terminfo**(4) manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

.bel=^G,

With this background information about specifying capabilities, let's add the capability string to our description of myterm. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

Basic Capabilities Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is ^G (**bel**).
- An 80-column wide screen (**cols**).
- A 30-line long screen (**lines**).
- Use of xon/xoff protocol (**xon**).

By combining the name string (see the section " Name the Terminal") and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities Screen-oriented capabilities manipulate the contents of a screen. Our example terminal myterm has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A <**CR**> is a CTRL-M (**cr**).
- A cursor up one line motion is a CTRL-K (**cuu1**).
- A cursor down one line motion is a CTRL-J (**cud1**).
- Moving the cursor to the left one space is a CTRL-H (**cub1**).
- Moving the cursor to the right one space is a CTRL-L (**cuf1**).
- Entering reverse video mode is an ESCAPE-D (**smso**).
- Exiting reverse video mode is an ESCAPE-Z (**rmso**).
- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).
- A terminal scrolls when receiving a <**NL**> at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
     am, bel=^G, cols#80, lines#30, xon,
     cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
     smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up arrow key generates an ESCAPE-[A (**kcuu1**).
- The down arrow key generates an ESCAPE-[B (**kcud1**).
- The right arrow key generates an ESCAPE-[C (**kcuf1**).
- The left arrow key generates an ESCAPE-[D (**kcub1**).
- The home key generates an ESCAPE-[H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
     am, bel=^G, cols#80, lines#30, xon,
     cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
     smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
     kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
     kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities Parameter string capabilities are capabilities that can take parameters — for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes (**sgr**) capabilities, are passed arguments in a **terminfo** program by the **tparm**() routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **printf**(3S) statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr**, takes nine arguments, one for each of the nine video attributes. See **terminfo**(4) for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence would be output.

Integer arguments are pushed onto the stack with a '%p' sequence followed by the argument number, such as '%p2' to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a '%d' sequence is used, exactly as in **printf**. Our terminal's **cup** sequence is built up as follows:

cup=	Meaning
\E[	output ESCAPE-[
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
        kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
        kcub1=\E[D, khome=\E[H,
        cup=\E[%i%p1%d;%p2%dH,
```

See **terminfo**(4) for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of myterm would be in a source file named **myterm.ti**. The compiled description of myterm would usually be placed in **/usr/lib/terminfo/m/myterm**, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm**, for example, to **/f/fancy**. If the environment variable **$TERMINFO** were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the **$TERMINFO** directory. All programs using the entry would then look in the new directory for the description file if **$TERMINFO** were set, before looking in the default **/usr/lib/terminfo**. The general format for the **tic** compiler is as follows:

> **tic** [**-v**] [**-c**] *file*

The **-v** option causes the compiler to trace its actions and output information about its progress. The **-c** option causes a check for errors; it may be combined with the **-v** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **cat**(1) to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

> **tic -v myterm.ti**<CR>
> (The trace information appears as the compilation proceeds.)

Refer to the **tic**(1M) manual page in the *System Administrator's Reference Manual* for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable **$TERMINFO** to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert line padding by commenting out **xon** in the description and then editing (using **vi**(1)) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput**(1) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the **tput** command is as follows:

> **tput** [-T*type*] *capname*

The type of terminal you are requesting information about is identified with the **-T***type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the " clear screen" character sequence for the terminal being used:

> **tput clear**
> (The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols
(The number of columns used by the terminal appears here.)
```

The **tput**(1) manual page found in the *User's Reference Manual* contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp**(1M) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420
```

compares the old and new 5420 entries.

To print out the **terminfo** source for the 5420, type

```
infocmp -I 5420
```

Converting a termcap Description to a terminfo Description

> *Caution: The* **terminfo** database is designed to take the place of the **termcap** database. Because of the many programs and processes that have been written with and for the **termcap** database, it is not feasible to do a complete cutover at one time. Any conversion from **termcap** to **terminfo** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **captoinfo**(1M) command converts **termcap**(4) descriptions to **terminfo**(4) descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example,

captoinfo /etc/termcap

converts the file **/etc/termcap** to **terminfo** source, preserving comments and other extraneous information within the file. The command line

captoinfo

looks up the current terminal in the **termcap** database, as specified by the **$TERM** and **$TERMCAP** environment variables and converts it to **terminfo**.

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp -C** to get the **termcap** descriptions.

If you have been using cursor optimization programs with the **-ltermcap** or **-ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **-lcurses** option.

curses Program Examples

The following examples demonstrate uses of **curses** routines.

The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move**(), **mvaddstr**(), **flash**(), **wnoutrefresh**() and **clrtoeol**() routines. These routines are all discussed in this chapter under " Working with **curses** Routines."

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch**() routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch**(), **delch**(), **insertln**(), and **deleteln**() routines. These functions insert and delete a character or line. See **curses**(3X) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

> *Note:* Because not all terminals have arrow keys, your **curses** programs will work on more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, another important point is that the input command is terminated by CTRL-D, not the escape key. It is very tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **curses** programs, avoid the escape key.

```
/* editor: A screen-oriented editor. The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

#include <stdio.h>
#include <curses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
    extern void perror(), exit();
        int i, n, l;
        int c;
        int line = 0;
        FILE *fd;
```

```
if (argc != 2)
{
        fprintf(stderr, " Usage: %s file\n" , argv[0]);
        exit(1);
}

fd = fopen(argv[1], " r" );
if (fd == NULL)
{
        perror(argv[1]);
        exit(2);
}

initscr();
cbreak();
nonl();
noecho();
idlok(stdscr, TRUE);
keypad(stdscr, TRUE);

/* Read in the file */
while ((c = getc(fd)) != EOF)
{
        if (c == '\n')
                line++;
        if (line > LINES - 2)
                break;
        addch(c);
}
fclose(fd);

move(0,0);
refresh();
edit();

/* Write out the file */
fd = fopen(argv[1], " w" );
for (l = 0; l < LINES - 1; l++)
```

```
	{
		n = len(l);
		for (i = 0; i < n; i++)
			putc(mvinch(l, i) & A_CHARTEXT, fd);
		putc('\n', fd);
	}
	fclose(fd);

	endwin();
	exit(0);
}

len(lineno)
int lineno;
{
	int linelen = COLS - 1;

	while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
		linelen--;
	return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
	int c;

	for (;;)
	{
		move(row, col);
		refresh();
		c = getch();

		/* Editor commands */
		switch (c)
		{
```

```
/* hjkl and arrow keys: move cursor
 * in direction indicated */
case 'h':
case KEY_LEFT:
        if (col > 0)
                col--;
        else
                flash();
        break;

case 'j':
case KEY_DOWN:
        if (row < LINES - 1)
                row++;
        else
                flash();
        break;

case 'k':
case KEY_UP:
        if (row > 0)
                row--;
        else
                flash();
        break;

case 'l':
case KEY_RIGHT:
        if (col < COLS - 1)
                col++;
        else
                flash();
        break;
/* i: enter input mode */
case KEY_IC:
case 'i':
        input();
        break;
```

```
                /* x: delete current character */
                case KEY_DC:
                case 'x':
                        delch();
                        break;

                /* o: open up a new line and enter input mode */
                case KEY_IL:
                case 'o':
                        move(++row, col = 0);
                        insertln();
                        input();
                        break;

                /* d: delete current line */
                case KEY_DL:
                case 'd':
                        deleteln();
                        break;

                /* ^L: redraw screen */
                case KEY_CLEAR:
                case CTRL('L'):
                        wrefresh(curscr);
                        break;

                /* w: write and quit */
                case 'w':
                        return;
                /* q: quit without writing */
                case 'q':
                        endwin();
                        exit(2);
                default:
                        flash();
                        break;
                }
        }
```

```
}

/*
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
        int c;

        standout();
        mvaddstr(LINES - 1, COLS - 20, " INPUT MODE" );
        standend();
        move(row, col);
        refresh();
        for (;;)
        {
                c = getch();
                if (c == CTRL('D') || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES - 1, COLS - 20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

The highlight Program

This program illustrates a use of the routine **attrset()**. **highlight** reads a text file and uses embedded escape sequences to control attributes. **\U** turns on underlining, **\B** turns on bold, and **\N** restores the default output attributes.

Note the first call to **scrollok()**, a routine that we have not previously discussed (see **curses**(3X)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **scrollok()** automatically scrolls the terminal up a line and calls **refresh()**.

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
        FILE *fd;
        int c, c2;
        void exit(), perror();

        if (argc != 2)
        {
                fprintf(stderr, " Usage: highlight file\n" );
                exit(1);
        }

        fd = fopen(argv[1], " r" );
```

```
        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        scrollok(stdscr, TRUE);
        nonl();
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\\')
                {
                        c2 = getc(fd);
                        switch (c2)
                        {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
                                attrset(0);
                                continue;
                        }
                        addch(c);
                        addch(c2);
                }
                else
                        addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

The scatter Program

This program takes the first **LINES - 1** lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *       The scatter program.
 */

#include        <curses.h>
#include        <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS]; /* Screen Array */
int  T[MAXLINES][MAXCOLS]; /* Tag Array - Keeps track of  *
                            * the number of characters    *
                            * printed and their positions. */

main()
{
        register int row = 0,col = 0;
        register int c;
        int char_count = 0;
        time_t t;
        void exit(), srand();

        initscr();
        for(row = 0;row < MAXLINES;row++)
                for(col = 0;col < MAXCOLS;col++)
                        s[row][col]=' ';

        col = row = 0;
```

```
    /* Read screen in */
    while ((c=getchar()) != EOF && row < LINES ) {

            if(c != '\n')
            {
                    /* Place char in screen array */
                    s[row][col++] = c;
                    if(c != ' ')
                            char_count++;
            }
            else
            {
                    col = 0;
                    row++;
            }
    }

    time(&t);       /* Seed the random number generator */
    srand((unsigned)t);

    while (char_count)
    {
            row = rand() % LINES;
            col = (rand() >> 2) % COLS;
            if (T[row][col] != 1 && s[row][col] != ' ')
            {
                    move(row, col);
                    addch(s[row][col]);
                    T[row][col] = 1;
                    char_count--;
                    refresh();
            }
    }
    endwin();
    exit(0);
}
```

The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak**() so that you can depress the space bar without having to hit return; it calls **noecho**() to prevent the space from echoing on the screen. The **nonl**() routine, which we have not previously discussed, is called to enable more cursor optimization. The **idlok**() routine, which we also have not discussed, is called to allow insert and delete line. (See **curses**(3X) for more information about these routines). Also notice that **clrtoeol**() and **clrtobot**() are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **curses**() program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ((fd=fopen(argv[1], "r")) == NULL)
        {
```

```
                perror(argv[1]);
                exit(2);
        }
        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
        idlok(stdscr, TRUE);

        while(1)
        {
                move(0,0);
                for (line = 0; line < LINES; line++)
                {
                        if (!fgets(linebuf, sizeof linebuf, fd))
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw(" %s" , linebuf);
                }
                refresh();
                if (getch() == 'q')
                        done();
        }
}

void done()
{
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command " **sleep 100000**" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
        void done(), exit();
        unsigned sleep();
        char *getenv();
        int c;

        if (argc != 4)
        {
            fprintf(stderr, " Usage: two othertty otherttytype inputfile\n" )
            exit(1);
```

```
}
fd = fopen(argv[3], " r" );
fdyou = fopen(argv[1], " w+" );
signal(SIGINT, done);   /* die gracefully */

me = newterm(getenv(" TERM" ), stdout, stdin);  /* initialize my tty */
you = newterm(argv[2], fdyou, fdyou);/* Initialize the other terminal */

set_term(me);    /* Set modes for my terminal */
noecho();        /* turn off tty echo */
cbreak();        /* enter cbreak mode */
nonl();          /* Allow linefeed */
nodelay(stdscr, TRUE); /* No hang on input */

set_term(you);   /* Set modes for other terminal */
noecho();
cbreak();
nonl();
nodelay(stdscr,TRUE);

/* Dump first screen full on my terminal */
dump_page(me);

/* Dump second screen full on the other terminal */
dump_page(you);

for (;;)  /* for each screen full */
{
    set_term(me);
    c = getch();
    if (c == 'q') /* wait for user to read it */
    done();
    if (c == ' ')
    dump_page(me);

    set_term(you);
    c = getch();
    if (c == 'q') /* wait for user to read it */
```

```
                done();
                if (c == ' ')
                dump_page(you);
                sleep(1);
        }
}
dump_page(term)
  SCREEN *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line = 0; line < LINES - 1; line++) {
            if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtobot();
            done();
            }
            mvaddstr(line, 0, linebuf);
        }
        standout();
        mvprintw(LINES - 1, 0, " --More--" );
        standend();
        refresh();          /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
        /* Clean up first terminal */
        set_term(you);
        move(LINES - 1,0);      /* to lower left corner */

        clrtoeol();         /* clear bottom line */
        refresh();          /* flush out everything */
        endwin();           /* curses cleanup */
```

```
        /* Clean up second terminal */
        set_term(me);
        move(LINES - 1,0);      /* to lower left corner */
        clrtoeol();         /* clear bottom line */
        refresh();          /* flush out everything */
        endwin();           /* curses cleanup */
        exit(0);
}
```

The window Program

This example program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh**() for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh**() on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin**() routine (which we have not discussed — see **curses**(3X)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin**() for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{

        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
        cbreak();

        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
                mvprintw(i, 0, " This is line %d of stdscr" , i);
```

```
        for (;;)

        {
                refresh();
                c = getch();
                switch (c)

                {

                case 'c': /* Enter command from keyboard */
                        werase(cmdwin);
                        wprintw(cmdwin, " Enter command:" );
                        wmove(cmdwin, 2, 0);
                        for (i = 0; i < COLS; i++)
                                waddch(cmdwin, '-');
                        wmove(cmdwin, 1, 0);
                        touchwin(cmdwin);
                        wrefresh(cmdwin);
                        wgetstr(cmdwin, buf);
                        touchwin(stdscr);

                        /*
                         * The command is now in buf.
                         * It should be processed here.
                         */

                case 'q':
                        endwin();
                        exit(0);
                }

        }

}
```

NAME

captoinfo - convert a termcap description into a terminfo description

SYNOPSIS

captoinfo [−**v** ...] [−**V**] [−**1**] [−**w** width] file ...

DESCRIPTION

captoinfo looks in *file* for *termcap* descriptions. For each one found, an equivalent *terminfo*(4) description is written to standard output, along with any comments found. A description which is expressed as relative to another description (as specified in the *termcap tc=* field) will be reduced to the minimum superset before being output.

If no *file* is given, then the environment variable **TERMCAP** is used for the filename or entry. If **TERMCAP** is a full pathname to a file, only the terminal whose name is specified in the environment variable **TERM** is extracted from that file. If the environment variable **TERMCAP** is not set, then the file */etc/termcap* is read.

−**v** print out tracing information on standard error as the program runs. Specifying additional −**v** options will cause more detailed information to be printed.

−**V** print out the version of the program in use on standard error and exit.

−**1** cause the fields to print out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters.

−**w** change the output to *width* characters.

FILES

/usr/lib/terminfo/?/* compiled terminal description database

CAVEATS

Certain *termcap* defaults are assumed to be true. For example, the bell character (*terminfo bel*) is assumed to be *^G*. The linefeed capability (*termcap nl*) is assumed to be the same for both *cursor_down* and *scroll_forward* (*terminfo cud1* and *ind*, respectively.) Padding information is assumed to belong at the end of the string.

The algorithm used to expand parameterized information for *termcap* fields such as *cursor_position* (*termcap cm*, *terminfo cup*) will sometimes produce a string which, though technically correct, may not be optimal. In particular, the rarely used *termcap* operation **%n** will produce strings that are especially long. Most occurrences of these non-optimal strings will be flagged with a warning message and may need to be recoded by hand.

The short two-letter name at the beginning of the list of names in a *termcap* entry, a hold-over from an earlier version of the UNIX system, has been removed.

DIAGNOSTICS

tgetent failed with return code n (reason).
: The termcap entry is not valid. In particular, check for an invalid 'tc=' entry.

unknown type given for the termcap code *cc*.
: The termcap description had an entry for *cc* whose type was not boolean, numeric or string.

wrong type given for the boolean (numeric, string) termcap code *cc*.
: The boolean *termcap* entry *cc* was entered as a numeric or string capability.

the boolean (numeric, string) termcap code *cc* is not a valid name.
: An unknown *termcap* code was specified.

tgetent failed on TERM=term.
: The terminal type specified could not be found in the *termcap* file.

TERM=term: **cap** *cc* (**info** *ii*) is NULL: REMOVED
: The *termcap* code was specified as a null string. The correct way to cancel an entry is with an '@', as in ':bs@:'. Giving a null string could cause incorrect assumptions to be made by the software which uses *termcap* or *terminfo*.

a function key for *cc* was specified, but it already has the value *vv*.
: When parsing the **ko** capability, the key *cc* was specified as having the same value as the capability *cc*, but the key *cc* already had a value assigned to it.

the unknown termcap name *cc* was specified in the **ko** termcap capability.
: A key was specified in the **ko** capability which could not be handled.

the *vi* character *v* (**info** *ii*) has the value *xx*, but **ma** gives *n*.
: The **ma** capability specified a function key with a value different from that specified in another setting of the same key.

the unknown *vi* key *v* was specified in the **ma** termcap capability.
: A *vi*(1) key unknown to *captoinfo* was specified in the **ma** capability.

Warning: *termcap* **sg** (*nn*) and *termcap* **ug** (*nn*) had different values.
: *terminfo* assumes that the **sg** (now **xmc**) and **ug** values were the same.

Warning: the string produced for *ii* may be inefficient.
: The parameterized string being created should be rewritten by hand.

Null termname given.
: The terminal type was null. This is given if the environment variable **TERM** is not set or is null.

cannot open *file* for reading.
: The specified file could not be opened.

SEE ALSO

infocmp(1M), tic(1M).
curses (3X), terminfo(4) in the *Programmer's Reference Manual.*
Chapter 10 in the *Programmer's Guide*.

NOTES

captoinfo should be used to convert *termcap* entries to *terminfo*(4) entries because the *termcap* database (from earlier versions of UNIX System V) may not be supplied in future releases.

NAME

curses - terminal screen handling and optimization package

SYNOPSIS

The *curses* manual page is organized as follows:

In SYNOPSIS
- compiling information
- summary of parameters used by *curses* routines
- alphabetical list of curses routines, showing their parameters

In DESCRIPTION:
- An overview of how *curses* routines should be used

In ROUTINES, descriptions of each *curses* routines, are grouped under the appropriate topics:
- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of **curscr**

Then come sections on:
- ATTRIBUTES
- FUNCTION CALLS
- LINE GRAPHICS

cc [flag ...] file ... **−lcurses** [library ...]

#include <curses.h> (automatically includes <**stdio.h**>, <**termio.h**>, and <**unctrl.h**>).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in <**curses.h**>.) Routines that return integers are not listed in the parameter list below.

bool bf

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp
char *cap, *capname, codename[2], erasechar, *filename, *fmt
char *keyname, killchar, *label, *longname
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ]
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ]
char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type

chtype attrs, ch, horch, vertch

FILE *infd, *outfd

int begin_x, begin_y, begline, bot, c, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fildes
int (*init( )), labfmt, labnum, line
int ms, ncols, new, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int p1, p2, p9, pmincol, pminrow, (*putc( )), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y

SCREEN *new, *newterm, *set_term

TERMINAL *cur_term, *nterm, *oterm

va_list varglist

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate( )
beep( )
box(win, vertch, horch)
cbreak( )
clear( )
clearok(win, bf)
clrtobot( )
clrtoeol( )
copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)"
curs_set(visibility)
def_prog_mode( )
def_shell_mode( )
del_curterm(oterm)
delay_output(ms)
delch( )
deleteln( )
delwin(win)
doupdate( )
draino(ms)
echo( )
echochar(ch)
endwin( )
erase( )
erasechar( )
filter( )
flash( )
flushinp( )
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)
getch( )

getmaxyx(win, y, x)
getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_ic( )
has_il( )
idlok(win, bf)
inch( )
initscr( )
insch(ch)
insertln( )
intrflush(win, bf)
isendwin( )
keyname(c)
keypad(win, bf)
killchar( )
leaveok(win, bf)
longname( )
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg...])
mvscanw(y, x, fmt [, arg...])
mvwaddch(win, y, x, ch)
mvwaddstr(win, y, x, str)
mvwdelch(win, y, x)
mvwgetch(win, y, x)
mvwgetstr(win, y, x, str)
mvwin(win, y, x)
mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)
nl( )
nocbreak( )
nodelay(win, bf)
noecho( )
nonl( )
noraw( )
notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)

pechochar(pad, ch)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow smaxcol)"
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, sma
printw(fmt [, arg...])
putp(str)
raw( )
refresh( )
reset_prog_mode( )
reset_shell_mode( )
resetty( )
restartterm(term, fildes, errret)
ripoffline(line, init)
savetty( )
scanw(fmt [, arg...])
scr_dump(filename)
scr_init(filename)
scr_restore(filename)
scroll(win)
scrollok(win, bf)
set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
slk_clear( )
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh( )
slk_refresh( )
slk_restore( )
slk_set(labnum, label, fmt)
slk_touch( )
standend( )
standout( )
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)
tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, ..., p9)
tputs(str, count, putc)
traceoff( )
traceon( )
typeahead(fildes)
unctrl(c)
ungetch(c)

vidattr(attrs)
vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)
waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)
wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg...])
wrefresh(win)
wscanw(win, fmt [, arg...])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

The *curses* routines give the user a terminal-independent method of updating screens with reasonable optimization.

In order to initialize the routines, the routine **initscr**( ) or **newterm**( ) must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine **endwin**( ) must be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented programs want this) after calling **initscr**( ) you should call **"cbreak**( ); **noecho**( );" Most programs would additionally call **"nonl**( ); **intrflush (stdscr, FALSE); keypad(stdscr, TRUE);"**.

Before a *curses* program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. For further details, see *profile*(4), *tput*(1), and the " Tabs and Initialization" subsection of *terminfo*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**( ).

Windows are referred to by variables declared as **WINDOW** *; the type **WINDOW** is defined in <**curses.h**> to be a C structure. These data structures are manipulated with routines described below, among which the most basic are **move**( ) and **addch**( ). (More general versions of these routines are included with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**.) Then **refresh**( ) is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of **newpad**( ) under " Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in <**curses.h**>, such as **A_REVERSE, ACS_HLINE,** and **KEY_LEFT.**

curses also defines the **WINDOW** * variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen in immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override **terminfo**'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* will check for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **att4425**, then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, *curses* will first check *$HOME/myterms/a/att4425*, and, if that fails, will then check */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in <**curses.h**>, and will be filled in by **initscr**( ) with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively. The constants **ERR** and **OK** are returned by routines to indicate whether the routine successfully completed. These constants are also defined in <**curses.h**>.

ROUTINES

Many of the following routines have two or more versions. The routines prefixed with **w** require a *window* argument. The routines prefixed with **p** require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv**( ) routines imply a call to **move**( ) before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always (**0**,**0**), not (**1**,**1**). The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**win** and **pad** are always of type **WINDOW** *.) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE.** (*bf* is always of type **bool.**) The types **WINDOW**, **bool**, and **chtype** are defined in <**curses.h**>. See the SYNOPSIS for a summary of what types all variables are.

All routines return either the integer **ERR** or the integer **OK**, unless otherwise noted. Routines that return pointers always return **NULL** on error.

Overall Screen Manipulation

WINDOW *initscr( )

The first routine called should almost always be **initscr**( ). (The exceptions are **slk_init**( ), **filter**( ), and **ripoffline**( ).) This will determine the terminal type and initialize all *curses* data structures. **initscr**( ) also arranges that the first call to **refresh**( ) will clear the screen. If errors occur, **initscr**( ) will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr** is returned. If the program wants an indication of error conditions, **newterm**( ) should be used instead of **initscr**( ). **initscr**( ) should only be called once per application.

endwin( )
A program should always call **endwin**( ) before exiting or escaping from *curses* mode temporarily, to do a shell escape or *system*(3S) call, for example. This routine will restore *tty*(7) modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **wrefresh**( ) or **doupdate**( ).

isendwin( )
Returns **TRUE** if **endwin**( ) has been called without any subsequent calls to **wrefresh**( ).

SCREEN *newterm(type, outfd, infd)
A program that outputs to more than one terminal must use **newterm**( ) for each terminal instead of **initscr**( ). A program that wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm**( ) should be called once for each terminal. It returns a variable of type **SCREEN*** that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin**( ) for each terminal being used. If **newterm**( ) is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin**( ) is called.

SCREEN *set_term(new)
This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates **SCREEN** pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh( )
wrefresh (win)
These routines (or **prefresh**( ), **pnoutrefresh**( ), **wnoutrefresh**( ), or **doupdate**( )) must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh**( ) copies the named window to the physical terminal screen, taking into account

what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh**( ) does the same thing, except it uses **stdscr** as a default window. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that **refresh**( ) is a macro.

wnoutrefresh(win)
doupdate( )
These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh**( ) alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copys the named window to the virtual screen, and then by calling **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) will result in alternating calls to **wnoutrefresh**( ) and **doupdate**( ), causing several bursts of output to the screen. By first calling **wnoutrefresh**( ) for each window, it is then possible to call **doupdate**( ) once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is **0**, they will be set to the value of **lines**−*begin_y* and **cols**−*begin_x*. A new full-screen window is created by calling **newwin(0,0,0,0)**.

mvwin(win, y, x)
Move the window so that the upper left corner will be at position (*y*, *x*). If the move would cause the window to be off the screen, it is an error and the window is not moved.

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y, begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **wrefresh**( ).

delwin(win) Delete the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

WINDOW *newpad(nlines, ncols)
Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**( ) with a pad as an argument; the routines **prefresh**( ) or **pnoutrefresh**( ) should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW *subpad(orig, nlines, ncols, begin_y, begin_x)
Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin**( ), which uses screen coordinates, the window is at position (*begin_y, begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **prefresh**( ).

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)" These routines are analogous to **wrefresh**( ) and **wnoutrefresh**( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow, smincol, smaxrow,* and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow, pmincol, sminrow,* or *smincol* are treated as if they were zero.

Output

These routines are used to "draw" text on windows.

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar* (see *putc*(3S)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok**( ) is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **clrtoeol**( ) before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the ^X notation. (Calling **winch**( ) after adding a control character will not return the control character, but instead will return the representation of the control character.)

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **inch**( ) and **addch**( ).) See **standout**( ), below.

Note that *ch* is actually of type **chtype**, not a character.

Note that **addch**( ), **mvaddch**( ), and **mvwaddch**( ), are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch) These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh**( ), a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**( ), the last location of the pad on the screen is reused for the arguments to **prefresh**( ).

Note that *ch* is actually of type **chtype**, not a character.

Note that **echochar**( ) is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str) These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch**( ) once for each character in the string.

Note that **addstr**( ), **mvaddstr**( ), and **mvwaddstr**( ) are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend( )
wstandend(win)
standout( )
wstandout(win) These routines manipulate the current attributes of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**. These constants are defined in <**curses.h**> and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**( ). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of the characters put on the screen.

attrset(attrs) sets the current attributes of the given window to *attrs*. **attroff**(attrs) turns off the named attributes without turning on or off any other attributes. **attron**(attrs) turns on the named attributes without affecting any others. **standout**( ) is the same as **attron(A_STANDOUT)**. **standend**( ) is the same as **attrset (0),** that is, it turns off all attributes.

Note that *attrs* is actually of type **chtype**, not a character.

Note that **attroff**( ), **attron**( ), **attrset**( ), **standend**( ), and **standout**( ) are macros.

beep( )
flash( )

These routines are used to signal the terminal user. **beep**( ) will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash**( ) will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

box(win, vertch, horch)

A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note that *vertch* and *horch* are actually of type **chtype**, not characters.

erase( )
werase(win)

These routines copy blanks to every position in the window.

Note that **erase**( ) is a macro.

clear( )
wclear(win)

These routines are like **erase**( ) and **werase**( ), but they also call **clearok**( ), arranging that the screen will be cleared completely on the next call to **wrefresh**( ) for that window, and repainted from scratch.

Note that **clear**( ) is a macro.

clrtobot( )
wclrtobot(win)

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot**( ) is a macro.

clrtoeol( )
wclrtoeol(win)

The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol**( ) is a macro.

delay_output(ms)

Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch( )
wdelch(win)
mvdelch(y, x)
mvwdelch(win, y, x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "delete-character" feature.)

Note that **delch**( ), **mvdelch**( ), and **mvwdelch**( ) are macros.

deleteln( )
wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware "delete-line" feature.)

Note that **deleteln**( ) is a macro.

getyx(win, y, x)

The cursor position of the window is placed in the two integer variables *y* and *x*. This is implemented as a macro, so no **"&"** is necessary before the variables.

getbegyx(win, y, x)
getmaxyx(win, y, x)

Like **getyx**( ), these routines store the current beginning coordinates and size of the specified window.

Note that **getbegyx**( ) and **getmaxyx**( ) are macros.

insch(ch)
winsch(win, ch)
mvwinsch(win, y, x, ch)
mvinsch(y, x, ch)

The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.)

Note that *ch* is actually of type **chtype**, not a character.

Note that **insch**( ), **mvinsch**( ), and **mvwinsch**( ) are macros.

insertln( )
winsertln(win)

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that **insertln**( ) is a macro.

move(y, x)
wmove(win, y, x)

The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **refresh**( ) is called. The position specified is relative to the upper left corner of the window, which is (**0**, **0**).

Note that **move**( ) is a macro.

overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)

These routines overlay *srcwin* on top of *dstwin*; that is, all text in *srcwin* is copied into *dstwin*. *scrwin* and *dstwin* need not be the same size; only text where the two windows overlap is copied. The difference is that **overlay**( ) is non-destructive (blanks are not copied), while **overwrite**( ) is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
This routine provides a finer grain of control over the **overlay**( ) and **overwrite**( ) routines. Like in the **prefresh**( ) routine, a rectangle is specified in the destination window, (*dminrow, dmincol*) and (*dmaxrow, dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow, smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay**( ).

printw(fmt [, arg...])
wprintw(win, fmt [, arg...])
mvprintw(y, x, fmt [, arg...])
mvwprintw(win, y, x, fmt [, arg...])
These routines are analogous to **printf**(3). The string which would be output by **printf**(3) is instead output using **waddstr**( ) on the given window.

vwprintw(win, fmt, varglist)
This routine corresponds to *vfprintf*(3S). It performs a **wprintw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

scroll(win)
The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

touchwin(win)
touchline(win, start, count)
Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that *count* lines have been changed, beginning with line *start* .

Input

getch( )
wgetch(win)
mvgetch(y, x)
mvwgetch(win, y, x)

A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value **ERR** is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window. No **refresh**( ) will occur between the **move**( ) and the **getch**( ) done within the routines **mvgetch**( ) and **mvwgetch**( ).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**( )) and ECHO mode (**echo**( )) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If **keypad**(win, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. (See **keypad**( ) under "Input Options Setting.") Possible function keys are defined in **<curses.h>** with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character routine is discouraged. Also see **notimeout**( ) below.)

Note that **getch**( ), **mvgetch**( ), and **mvwgetch**( ) are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **getch**( ) is made, until a newline, carriage return, or enter key is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. As in **mvgetch**( ), no **refresh**( ) is done between the **move**( ) and **getstr**( ) within the routines **mvgetstr**( ) and **mvwgetstr**( ).

Note that **getstr**( ), **mvgetstr**( ), and **mvwgetstr**( ) are macros.

flushinp( )

Throws away any typeahead that has been typed by the user and has not yet been read by the program.

ungetch(c)

Place *c* back onto the input queue to be returned by the next call to **wgetch**( ).

inch( )
winch(win)
mvinch(y, x)
mvwinch(win, y, x)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES,** defined in <**curses.h**>, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that **inch**( ), **winch**( ), **mvinch**( ), and **mvwinch**( ) are macros.

scanw(fmt [, arg...])
wscanw(win, fmt [, arg...])
mvscanw(y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])

These routines correspond to *scanf*(3S), as do their arguments and return values. **wgetstr**( ) is called on the window, and the resulting line is used as input for the scan.

vwscanw(win, fmt, ap)

This routine is similar to **vwprintw**( ) above in that performs a **wscanw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <**varargs.h**>. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**( ).

clearok(win, bf) If enabled (*bf* is **TRUE**), the next call to **wrefresh**( ) with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf) If enabled (*bf* is **TRUE**), *curses* will consider using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is FALSE), *curses* will very seldom use this feature. (The "insert/delete-character" feature is always considered.) This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it isn't really needed. If "insert/delete-line" cannot be used, *curses* will redraw the changed portions of all lines.

leaveok(win, bf) Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

setscrreg(top, bot)
wsetscrreg(win, top, bot)
These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok**( ) are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok**( ) is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.)

Note that **setscrreg**( ) and **wsetscrreg**( ) are macros.

scrollok(win, bf) This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is **FALSE**), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh**( ) is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**( ).)

nl( )
nonl( ) These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using **nonl**( ), *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling **endwin**( ).

For more information on these options, see Chapter 10 of the *Programmer's Guide*.

cbreak( )
nocbreak( ) These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see *termio*(7)). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak**( ) or **nocbreak**( ) explicitly. Most interactive programs using *curses* will set CBREAK mode.

Note that **cbreak**( ) overrides **raw**( ). See **getch**( ) under " Input" for a discussion of how these routines interact with **echo**( ) and **noecho**( ).

echo( )
noecho( )

These routines control whether characters typed by the user are echoed by **getch**( ) as they are typed. Echoing by the tty driver is always disabled, but initially **getch**( ) is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho**( ). See **getch**( ) under " Input" for a discussion of how these routines interact with **cbreak**( ) and **nocbreak**( ).

halfdelay(tenths)

Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed. *tenths* must be a number between 1 and 255. Use **nocbreak**( ) to leave half-delay mode.

intrflush(win, bf)

If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad(win, bf)

This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch**( ) will return a single value representing the function key, as in **KEY_LEFT**. If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when **wgetch**( ) is called.

meta(win, bf)

If enabled, characters returned by **wgetch**( ) are transmitted with all 8 bits, instead of with the highest bit stripped. In order for **meta**( ) to work correctly, the **km** (has_meta_key) capability has to be specified in the terminal's **terminfo**(4) entry.

nodelay(win, bf) This option causes **wgetch**( ) to be a non-blocking call. If no input is ready, **wgetch**( ) will return ERR. If disabled, **wgetch**( ) will hang until a key is pressed.

notimeout(win, bf) While interpreting an input escape sequence, **wgetch**( ) will set a timer while waiting for the next character. If **notimeout**(win, **TRUE**) is called, then **wgetch**( ) will not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

raw( )
noraw( ) The terminal is placed into or out of raw mode. RAW mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal. RAW mode also causes 8-bit input and output. The behavior of the BREAK key depends on other bits in the *tty*(7) driver that are not set by *curses*.

typeahead(fildes) *curses* does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **refresh**( ) or **doupdate**( ) is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm**( ), or **stdin** in the case that **initscr**( ) was used, will be used to do this typeahead checking. The **typeahead**( ) routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **−1,** then no typeahead checking will be done.

Note that *fildes* is a file descriptor, not a <**stdio.h**> FILE pointer.

Environment Queries

baudrate( ) Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar( ) The user's current erase character is returned.

has_ic( ) True if the terminal has insert- and delete-character capabilities.

has_il( ) True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok**( ).

char killchar( ) The user's current line-kill character is returned.

char *longname( ) This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr**( ) or **newterm**( ). The area is overwritten by each call to **newterm**( ) and is not restored by **set_term**( ), so the value should be saved between calls to **newterm**( ) if **longname**( ) is going to be used with multiple terminals.

Soft Labels

If desired, *curses* will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, *curses* will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. *curses* standardizes on 8 labels of 8 characters each.

slk_init(labfmt) In order to use soft labels, this routine must be called before **initscr**( ) or **newterm**( ) is called. If **initscr**( ) winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set(labnum, label, labfmt)
labnum is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A **NULL** string or a **NULL** pointer will put up a blank label. *labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh( )
slk_noutrefresh( ) These routines correspond to the routines **wrefresh**( ) and **wnoutrefresh**( ). Most applications would use **slk_noutrefresh**( ) because a **wrefresh**( ) will most likely soon follow.

char *slk_label(labnum)
: The current label for label number *labnum*, with leading and trailing blanks stripped, is returned.

slk_clear( )
: The soft labels are cleared from the screen.

slk_restore( )
: The soft labels are restored to the screen after a **slk_clear**( ).

slk_touch( )
: All of the soft labels are forced to be output the next time a **slk_noutrefresh**( ) is performed.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of library routines.

def_prog_mode( )
def_shell_mode( )
: Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode**( ) and **reset_shell_mode**( ) routines. This is done automatically by **initscr**( ).

reset_prog_mode( )
reset_shell_mode( )
: Restore the terminal to "program" (in **curses**) or "shell" (out of *curses*) state. These are done automatically by **endwin**( ) and **doupdate**( ) after an **endwin**( ), so they normally would not be called.

resetty( )
savetty( )
: These routines save and restore the state of the terminal modes. **savetty**( ) saves the current state of the terminal in a buffer and **resetty**( ) restores the state to what it was at the last call to **savetty**( ).

getsyx(y, x)
: The current coordinates of the virtual screen cursor are returned in *y* and *x*. Like **getyx**( ), the variables *y* and *x* do not take an "**&**" before them. If **leaveok**( ) is currently **TRUE**, then **−1**,**−1** will be returned. If lines may have been removed from the top of the screen using **ripoffline**( ) and the values are to be used beyond just passing them on to **setsyx**( ), the value *y*+**stdscr->_yoffset** should be used for those other uses.

: Note that **getsyx**( ) is a macro.

setsyx(y, x)
The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **−1**, then **leaveok**( ) will be set. The two routines **getsyx**( ) and **setsyx**( ) are designed to be used by a library routine which manipulates curses windows but does not want to mess up the current position of the program's cursor. The library routine would call **getsyx**( ) at the beginning, do its manipulation of its own windows, do a **wnoutrefresh**( ) on its windows, call **setsyx**( ), and then call **doupdate**( ).

ripoffline(line, init)
This routine provides access to the same facility that **slk_init**( ) uses to reduce the size of the screen. **ripoffline**( ) must be called before **initscr**( ) or **newterm**( ) is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr**( ), the routine *init*( ) is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in <**curses.h**>) are not guaranteed to be accurate and **wrefresh**( ) or **doupdate**( ) must not be called. It is allowable to call **wnoutrefresh**( ) during the initialization routine.

ripoffline( ) can be called up to five times before calling **initscr**( ) or **newterm**( ).

scr_dump(filename)
The current contents of the virtual screen are written to the file *filename*.

scr_restore(filename)
The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**( ). The next call to **doupdate**( ) will restore the screen to what it looked like in the dump file.

scr_init(filename)
The contents of *filename* are read in and used to initialize the *curses* data structures about what the terminal currently has on its screen. If the data is determined to be valid, *curses* will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**( ) would be used after **initscr**( ) or a *system*(3S) call to share the screen with another process which has done a

scr_dump( ) after its **endwin**( ) call. The data will be declared invalid if the time-stamp of the tty is old or the *terminfo*(4) capability **nrrmc** is true.

curs_set(visibility) The cursor is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**.

draino(ms) Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

garbagedlines(win, begline, numlines)
This routine indicates to *curses* that a screen line is garbaged and should be thrown away before having anything written over the top of it. It could be used for programs such as editors which want a command to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire screen would be subject to even more communication noise. Just redrawing the single line gives some semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

napms(ms) Sleep for *ms* milliseconds.

Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm**( ) should be called. (Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).) This will define the set of terminal-dependent variables defined in the *terminfo*(4) database. The *terminfo*(4) variables **lines** and **columns** (see *terminfo*(4)) are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist and the program is running in a layer (see *layers*(1)), the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files <**curses.h**> and <**term.h**> should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo*(4) strings (including the output of **tparm**( )) should be printed with **tputs**( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo*(4)). (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the *curses* routines (see **endwin**( )).

setupterm(term, fildes, errret)
: Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*; if *term* is **NULL**, the environment variable **TERM** will be used. All output is to the file descriptor *fildes*. If *errret* is not **NULL**, then **setupterm**( ) will return **OK** or **ERR** and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **−1** means that the *terminfo*(4) database could not be found. If *errret* is **NULL, setupterm**( ) will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type **TERMINAL**. After **setupterm**( ) returns successfully, the variable **cur_term** (of type **TERMINAL** *) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm**( ) again. Further calls to **setupterm**( ) will allocate new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm)
: *nterm* is of type **TERMINAL** *. **set_curterm**( ) sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm)
: *oterm* is of type **TERMINAL** *. **del_curterm**( ) frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm**( ) has been called.

restartterm(term, fildes, errret)
: Like **setupterm**( ) after a memory restore.

char ***tparm**(str, p_1, p_2, ..., p_9)
: Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc)
: Apply padding to the string *str* and output it. *str* must be a *terminfo*(4) string variable or the return value from **tparm**( ), **tgetstr**( ), **tigetstr**( ) or **tgoto**( ). *count* is the number of lines affected, or **1** if not applicable. *putc*( ) is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

putp(str)
: A routine that calls **tputs** (*str*, **1**, **putchar**( )).

vidputs(attrs, putc)
: Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*( ).

vidattr(attrs)
: Like **vidputs**( ), except that it outputs through *putchar*(3S).

mvcur(oldrow, oldcol, newrow, newcol)
: Low-level cursor motion.

The following routines return the value of the capability corresponding to the *terminfo*(4) *capname* passed to them, such as **xenl**.

tigetflag(capname)
: The value **−1** is returned if *capname* is not a boolean capability.

tigetnum(capname)
: The value **−2** is returned if *capname* is not a numeric capability.

tigetstr(capname)
: The value (char *) **−1** is returned if *capname* is not a string capability.

char ***boolnames**[ ], ***boolcodes**[ ], ***boolfnames**[ ]
char ***numnames**[ ], ***numcodes**[ ], ***numfnames**[ ]
char ***strnames**[ ], ***strcodes**[ ], ***strfnames**[ ]
: These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo*(4) variables.

Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

tgetent(bp, name) Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.

tgetflag(codename) Get the boolean entry for *codename*.

tgetnum(codes) Get numeric entry for *codename*.

char *tgetstr(codename, area)
Return the string entry for *codename*. If *area* is not **NULL**, then also store it in the buffer pointed to by *area* and advance *area*. **tputs**( ) should be used to output the returned string.

char *tgoto(cap, col, row)
Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**( ).

tputs(str, affcnt, putc)
See **tputs**( ) above, under "Terminfo-Level Manipulations".

Miscellaneous

traceoff( )
traceon( ) Turn off and on debugging trace output when using the debug version of the *curses* library, */usr/lib/libdcurses.a*. This facility is available only to customers with a source license.

unctrl(c) This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

unctrl( ) is a macro, defined in <**unctrl.h**>, which is automatically included by <**curses.h**>.

char *keyname(c) A character string corresponding to the key *c* is returned.

filter( ) This routine is one of the few that is to be called before **initscr**( ) or **newterm**( ) is called. It arranges things so that *curses* thinks that there is a 1-line screen. *curses* will not use any terminal capabilities that assume that they know what line on the screen the cursor is on.

Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. (This is how most programs would implement a "repaint-screen" routine.) The source window argument to **overlay**( ), **overwrite**( ), and **copywin**( ) may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode( )	Replaced by **cbreak**( ).
fixterm( )	Replaced by **reset_prog_mode**( ).
gettmode( )	A no-op.
nocrmode( )	Replaced by **nocbreak**( ).
resetterm( )	Replaced by **reset_shell_mode**( ).
saveterm( )	Replaced by **def_prog_mode**( ).
setterm( )	Replaced by **setupterm**( ).

ATTRIBUTES

The following video attributes, defined in <**curses.h**>, can be passed to the routines **attron**( ), **attroff**( ), and **attrset**( ), or OR'ed with the characters passed to **addch**( ).

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_ALTCHARSET	Alternate character set
A_CHARTEXT	Bit-mask to extract character (described under **winch**( ))
A_ATTRIBUTES	Bit-mask to extract attributes (described under **winch**( ))
A_NORMAL	Bit mask to reset all attributes off (for example: **attrset (A_NORMAL)**

FUNCTION-KEYS

The following function keys, defined in <**curses.h**>, might be returned by **getch**() if **keypad**() has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo*(4) database.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for f_n.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	soft (partial) reset
KEY_RESET	0531	reset or hard reset
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left) keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	beg(inning) key
KEY_CANCEL	0543	cancel key
KEY_CLOSE	0544	close key
KEY_COMMAND	0545	cmd (command) key

KEY_COPY	0546	copy key
KEY_CREATE	0547	create key
KEY_END	0550	end key
KEY_EXIT	0551	exit key
KEY_FIND	0552	find key
KEY_HELP	0553	help key
KEY_MARK	0554	mark key
KEY_MESSAGE	0555	message key
KEY_MOVE	0556	move key
KEY_NEXT	0557	next object key
KEY_OPEN	0560	open key
KEY_OPTIONS	0561	options key
KEY_PREVIOUS	0562	previous object key
KEY_REDO	0563	redo key
KEY_REFERENCE	0564	ref(erence) key
KEY_REFRESH	0565	refresh key
KEY_REPLACE	0566	replace key
KEY_RESTART	0567	restart key
KEY_RESUME	0570	resume key
KEY_SAVE	0571	save key
KEY_SBEG	0572	shifted beginning key
KEY_SCANCEL	0573	shifted cancel key
KEY_SCOMMAND	0574	shifted command key
KEY_SCOPY	0575	shifted copy key
KEY_SCREATE	0576	shifted create key
KEY_SDC	0577	shifted delete char key
KEY_SDL	0600	shifted delete line key
KEY_SELECT	0601	select key
KEY_SEND	0602	shifted end key
KEY_SEOL	0603	shifted clear line key
KEY_SEXIT	0604	shifted exit key
KEY_SFIND	0605	shifted find key
KEY_SHELP	0606	shifted help key
KEY_SHOME	0607	shifted home key
KEY_SIC	0610	shifted input key
KEY_SLEFT	0611	shifted left arrow key
KEY_SMESSAGE	0612	shifted message key
KEY_SMOVE	0613	shifted move key
KEY_SNEXT	0614	shifted next key
KEY_SOPTIONS	0615	shifted options key
KEY_SPREVIOUS	0616	shifted prev key
KEY_SPRINT	0617	shifted print key
KEY_SREDO	0620	shifted redo key
KEY_SREPLACE	0621	shifted replace key
KEY_SRIGHT	0622	shifted right arrow
KEY_SRSUME	0623	shifted resume key
KEY_SSAVE	0624	shifted save key
KEY_SSUSPEND	0625	shifted suspend key
KEY_SUNDO	0626	shifted undo key
KEY_SUSPEND	0627	suspend key
KEY_UNDO	0630	undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch**( ). When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default charcter listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	-	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	-	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

RETURN VALUES

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **setscrreg**( ), **wsetscrreg**( ), **getsyx**( ), **getyx**( ), **getbegy**( ), **getmaxyx**( ). For these macros, no useful value is returned.

Routines that return pointers always return **(type *) NULL** on error.

BUGS

Currently typeahead checking is done using a nodelay read followed by an **ungetch**( ) of any character that may have been read. Typeahead checking is done only if **wgetch**( ) has been called at least once. This will be changed when proper kernel support is available. Programs which use a mixture of their own input routines with *curses* input routines may wish to call **typeahead(−1)** to turn off typeahead checking.

The argument to **napms**( ) is currently rounded up to the nearest second.

draino (ms) only works for *ms* equal to **0**.

WARNINGS

To use the new *curses* features, use the Release 3.0 version of *curses* on UNIX System Release 3.0. All programs that ran with System V Release 2 *curses* will run with System V Release 3.0. You may link applications with object files based on the Release 2 *curses/terminfo* with the Release 3.0 *libcurses.a* library. You may link applications with object files based on the Release 3.0 *curses/terminfo* with the Release 2 *libcurses.a* library, so long as the application does not use the new features in the Release 3.0 *curses/terminfo.*

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names **erase**( ) and **move**( ). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or **#undef move**( ) and **erase( )** in the *plot*(3X) code.

Between the time a call to **initscr**( ) and **endwin**( ) has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" (see *stdio*(3S)) for output during that time can cause unpredictable results.

SEE ALSO

cc(1), ld(1), ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S), profile(4), term(4), terminfo(4), varargs(5).
termio(7), tty(7) in the *System Administrator's Reference Manual.*
Chapter 10 of the *Programmer's Guide*.

NAME

infocmp - compare or print out terminfo descriptions

SYNOPSIS

infocmp [**−d**] [**−c**] [**−n**] [**−I**] [**−L**] [**−C**] [**−r**] [**−u**] [**−s d|i|l|c**] [**−v**] [**−V**] [**−1**] [**−w** width] [**−A** directory] [**−B** directory] [termname ...]

DESCRIPTION

infocmp can be used to compare a binary *terminfo*(4) entry with other terminfo entries, rewrite a *terminfo*(4) description to take advantage of the **use=** terminfo field, or print out a *terminfo*(4) description from the binary file (*term*(4)) in a variety of formats. In all cases, the boolean fields will be printed first, followed by the numeric fields, followed by the string fields.

Default Options

If no options are specified and zero or one *termnames* are specified, the **−I** option will be assumed. If more than one *termname* is specified, the **−d** option will be assumed.

Comparison Options [-d] [-c] [-n]

infocmp compares the *terminfo*(4) description of the first terminal *termname* with each of the descriptions given by the entries for the other terminal's *termnames*. If a capability is defined for only one of the terminals, the value returned will depend on the type of the capability: **F** for boolean variables, **−1** for integer variables, and **NULL** for string variables.

−d produce a list of each capability that is different. In this manner, if one has two entries for the same terminal or similar terminals, using *infocmp* will show what is different between the two entries. This is sometimes necessary when more than one person produces an entry for the same terminal and one wants to see what is different between the two.

−c produce a list of each capability that is common between the two entries. Capabilities that are not set are ignored. This option can be used as a quick check to see if the **−u** option is worth using.

−n produce a list of each capability that is in neither entry. If no *termnames* are given, the environment variable **TERM** will be used for both of the *termnames*. This can be used as a quick check to see if anything was left out of the description.

Source Listing Options [-I] [-L] [-C] [-r]

The **−I**, **−L**, and **−C** options will produce a source listing for each terminal named.

−I use the *terminfo*(4) names

−L use the long C variable name listed in <**term.h**>

−C use the *termcap* names

−r when using **−C**, put out all capabilities in *termcap* form

If no *termnames* are given, the environment variable **TERM** will be used for the terminal name.

The source produced by the **−C** option may be used directly as a *termcap* entry, but not all of the parameterized strings may be changed to the *termcap* format. *infocmp* will attempt to convert most of the parameterized information, but that which it doesn't will be plainly marked in the output and commented out. These should be edited by hand.

All padding information for strings will be collected together and placed at the beginning of the string where *termcap* expects it. Mandatory padding (padding information with a trailing '/') will become optional.

All *termcap* variables no longer supported by *terminfo*(4), but which are derivable from other *terminfo*(4) variables, will be output. Not all *terminfo*(4) capabilities will be translated; only those variables which were part of *termcap* will normally be output. Specifying the **−r** option will take off this restriction, allowing all capabilities to be output in *termcap* form.

Note that because padding is collected to the beginning of the capability, not all capabilities are output, mandatory padding is not supported, and *termcap* strings were not as flexible, it is not always possible to convert a *terminfo*(4) string capability into an equivalent *termcap* format. Not all of these strings will be able to be converted. A subsequent conversion of the *termcap* file back into *terminfo*(4) format will not necessarily reproduce the original *terminfo*(4) source.

Some common *terminfo* parameter sequences, their *termcap* equivalents, and some terminal types which commonly have such sequences, are:

Terminfo	Termcap	Representative Termina
%p1%c	%.	adm
%p1%d	%d	hp, ANSI standard, vt10
%p1%'x'%+%c	%+x	concept
%i	%i	ANSI standard, vt100
%p1%?%'x'%>%t%p1%'y'%+%;	%>xy	concept
%p2 is printed before %p1	%r	hp

Use= Option [-u]

−u produce a *terminfo*(4) source description of the first terminal *termname* which is relative to the sum of the descriptions given by the entries for the other terminals *termnames*. It does this by analyzing the differences between the first *termname* and the other *termnames* and producing a description with **use=** fields for the other terminals. In this manner, it is possible to retrofit generic terminfo entries into a terminal's description. Or, if two similar terminals exist, but were coded at different times or by different people so that each description is a full description, using *infocmp* will show what can be done to change one description to be relative to the other.

A capability will get printed with an at-sign (@) if it no longer exists in the first *termname*, but one of the other *termname* entries contains a value for it. A capability's value gets printed if the value in the first *termname* is not found in any of the other *termname* entries, or if the first of the other *termname* entries that has this capability gives a different value for the capability than that in the first *termname*.

The order of the other *termname* entries is significant. Since the terminfo compiler **tic**(1M) does a left-to-right scan of the capabilities, specifying two **use**= entries that contain differing entries for the same capabilities will produce different results depending on the order that the entries are given in. *infocmp* will flag any such inconsistencies between the other *termname* entries as they are found.

Alternatively, specifying a capability *after* a **use**= entry that contains that capability will cause the second specification to be ignored. Using *infocmp* to recreate a description can be a useful check to make sure that everything was specified correctly in the original source description.

Another error that does not cause incorrect compiled files, but will slow down the compilation time, is specifying extra **use**= fields that are superfluous. *infocmp* will flag any other *termname* **use**= fields that were not needed.

Other Options [-s d|i|l|c] [-v] [-V] [-1] [-w width]

−s sort the fields within each type according to the argument below:

- **d** leave fields in the order that they are stored in the *terminfo* database.
- **i** sort by *terminfo* name.
- **l** sort by the long C variable name.
- **c** sort by the *termcap* name.

If no **−s** option is given, the fields printed out will be sorted alphabetically by the *terminfo* name within each type, except in the case of the **−C** or the **−L** options, which cause the sorting to be done by the *termcap* name or the long C variable name, respectively.

−v print out tracing information on standard error as the program runs.

−V print out the version of the program in use on standard error and exit.

−1 cause the fields to printed out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters.

−w change the output to width characters.

Changing Databases [-A directory] [-B directory]

The location of the compiled *terminfo*(4) database is taken from the environment variable **TERMINFO**. If the variable is not defined, or the terminal is not found in that location, the system *terminfo*(4) database, usually in */usr/lib/terminfo*, will be used. The options −**A** and −**B** may be used to override this location. The −**A** option will set **TERMINFO** for the first *termname* and the −**B** option will set **TERMINFO** for the other *termnames*. With this, it is possible to compare descriptions for a terminal with the same name located in two different databases. This is useful for comparing descriptions for the same terminal created by different people. Otherwise the terminals would have to be named differently in the *terminfo*(4) database for a comparison to be made.

FILES

/usr/lib/terminfo/?/* compiled terminal description database

DIAGNOSTICS

malloc is out of space!
: There was not enough memory available to process all the terminal descriptions requested. Run *infocmp* several times, each time including a subset of the desired *termnames*.

use= order dependency found:
: A value specified in one relative terminal specification was different from that in another relative terminal specification.

'use=*term*' did not add anything to the description.
: A relative terminal name did not contribute anything to the final description.

must have at least two terminal names for a comparison to be done.
: The −**u**, −**d** and −**c** options require at least two terminal names.

SEE ALSO

tic(1M), curses(3X), term(4), terminfo(4) in the *Programmer's Reference Manual*.
captoinfo(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide*.

NOTE

The *termcap* database (from earlier releases of UNIX System V) may not be supplied in future releases.

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

terminfo is a compiled database (see *tic*(1M)) describing the capabilities of terminals. Terminals are described in *terminfo* source descriptions by giving a set of capabilities which they have, by describing how operations are performed, by describing padding requirements, and by specifying initialization sequences. This database is used by applications programs, such as *vi*(1) and *curses*(3X), so they can work with a variety of terminals without changes to the programs. To obtain the source description for a terminal, use the **–I** option of *infocmp*(1M).

Entries in *terminfo* source files consist of a number of comma-separated fields. White space after each comma is ignored. The first line of each terminal description in the *terminfo* database gives the name by which *terminfo* knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERM** in *$HOME/.profile*; see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. See *term*(5) for examples and more information on choosing names and synonyms.

CAPABILITIES

In the table below, the **Variable** is the name by which the C programmer (at the *terminfo* level) accesses the capability. The **Capname** is the short name for this variable used in the text of the database. It is used by a person updating the database and by the *tput*(1) command when asking what the value of the capability is for a particular terminal. The **Termcap Code** is a two-letter code that corresponds to the old *termcap* capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the table below, have names beginning with **key_**. The following indicators may appear at the end of the **Description** for a variable.

(G) indicates that the string is passed through **tparm**( ) with parameters (parms) as given ($\#_i$).

(*) indicates that padding may be based on the number of lines affected.

($\#_i$) indicates the i^{th} parameter.

Variable	Cap-name	Termcap Code	Description
Booleans:			
auto_left_margin	bw	bw	**cub1** wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
no_esc_ctlc	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch	xenl	xn	Newline ignored after 80 cols (*Conce*
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g. dialup, switch
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see.
has_meta_key	km	km	Has a meta key (shift, sets parity bi
has_status_line	hs	hs	Has extra " status line"
insert_null_glitch	in	in	Insert mode distinguishes nulls
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff requir
non_rev_rmcup	nrrmc	NR	**smcup** does not reverse **rmcup**
no_pad_char	npc	NP	Pad character doesn't exist
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen.
status_line_esc_ok	eslok	es	Escape can be used on the status line
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic **smso** char (t1061)
tilde_glitch	hz	hz	Hazeltine; can't print tildes(~)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns	cols	co	Number of columns in a line
init_tabs	it	it	Tabs initially every # spaces.
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of cols in each label
lines	lines	li	Number of lines on screen or page
lines_of_memory	lm	lm	Lines of memory if > **lines**; **0** means varies
magic_cookie_glitch	xmc	sg	Number blank chars left by **smso** or **rmso**

num_labels	nlab	Nl	Number of labels on screen (start at 1)
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number (UNIX system)
width_status_line	wsl	ws	Number of columns in status line

Strings:

acs_chars	acsc	ac	Graphic charset pairs aAbBcC - def=vt100+
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return (*)
change_scroll_region	csr	cs	Change to lines #1 thru #2 (vt100) (G)
char_padding	rmp	rP	Like **ip** but when in replace mode
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear left and right soft margins
clear_screen	clear	cl	Clear screen and home cursor (*)
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display (*)
column_address	hpa	ch	Horizontal position absolute (G)
command_character	cmdch	CC	Term. settable cmd char in prototype
cursor_address	cup	cm	Cursor motion to row #1 col #2 (G)
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space.
cursor_mem_address	mrcup	CM	Memory relative cursor addressing (G)
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
delete_character	dch1	dc	Delete character (*)
delete_line	dl1	dl	Delete line (*)
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate char set
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_insert_mode	smir	im	Insert mode (enter);
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode	smso	so	Begin standout mode
enter_underline_mode	smul	us	Start underscore mode
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking

erase_chars	ech	ec	Erase #1 characters (G)
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use **cup**
exit_delete_mode	rmdc	ed	End delete mode
exit_insert_mode	rmir	ei	End insert mode;
exit_standout_mode	rmso	se	End standout mode
exit_underline_mode	rmul	ue	End underscore mode
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject (*)
from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal initialization string
init_2string	is2	is	Terminal initialization string
init_3string	is3	i3	Terminal initialization string
init_file	if	if	Name of initialization file containing **is**
init_prog	iprog	iP	Path name of program for init.
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line (*)
insert_padding	ip	ip	Insert pad after character inserted (*)
key_a1	ka1	K1	KEY_A1, 0534, Upper left of keypad
key_a3	ka3	K3	KEY_A3, 0535, Upper right of keypad
key_b2	kb2	K2	KEY_B2, 0536, Center of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, 0407, Sent by backspace key
key_beg	kbeg	@1	KEY_BEG, 0542, Sent by beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, 0541, Sent by back-tab key
key_c1	kc1	K4	KEY_C1, 0537, Lower left of keypad
key_c3	kc3	K5	KEY_C3, 0540, Lower right of keypad
key_cancel	kcan	@2	KEY_CANCEL, 0543, Sent by cancel key
key_catab	ktbc	ka	KEY_CATAB, 0526, Sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, 0515, Sent by clear-screen or erase key
key_close	kclo	@3	KEY_CLOSE, 0544, Sent by close key
key_command	kcmd	@4	KEY_COMMAND, 0545, Sent by cmd (command) key
key_copy	kcpy	@5	KEY_COPY, 0546, Sent by copy key
key_create	kcrt	@6	KEY_CREATE, 0547, Sent by create key
key_ctab	kctab	kt	KEY_CTAB, 0525, Sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, 0512, Sent by delete-character key
key_dl	kdl1	kL	KEY_DL, 0510, Sent by delete-line key
key_down	kcud1	kd	KEY_DOWN, 0402, Sent by terminal down-arrow key
key_eic	krmir	kM	KEY_EIC, 0514, Sent by **rmir** or **smir** in insert mode
key_end	kend	@7	KEY_END, 0550, Sent by end key

key_enter	kent	@8	KEY_ENTER, 0527, Sent by enter/send key
key_eol	kel	kE	KEY_EOL, 0517, Sent by clear-to-end-of-line key
key_eos	ked	kS	KEY_EOS, 0516, Sent by clear-to-end-of-screen key
key_exit	kext	@9	KEY_EXIT, 0551, Sent by exit key
key_f0	kf0	k0	KEY_F(0), 0410, Sent by function key f0
key_f1	kf1	k1	KEY_F(1), 0411, Sent by function key f1
key_f2	kf2	k2	KEY_F(2), 0412, Sent by function key f2
key_f3	kf3	k3	KEY_F(3), 0413, Sent by function key f3
key_f4	kf4	k4	KEY_F(4), 0414, Sent by function key f4
key_f5	kf5	k5	KEY_F(5), 0415, Sent by function key f5
key_f6	kf6	k6	KEY_F(6), 0416, Sent by function key f6
key_f7	kf7	k7	KEY_F(7), 0417, Sent by function key f7
key_f8	kf8	k8	KEY_F(8), 0420, Sent by function key f8
key_f9	kf9	k9	KEY_F(9), 0421, Sent by function key f9
key_f10	kf10	k;	KEY_F(10), 0422, Sent by function key f10
key_f11	kf11	F1	KEY_F(11), 0423, Sent by function key f11
key_f12	kf12	F2	KEY_F(12), 0424, Sent by function key f12
key_f13	kf13	F3	KEY_F(13), 0425, Sent by function key f13
key_f14	kf14	F4	KEY_F(14), 0426, Sent by function key f14
key_f15	kf15	F5	KEY_F(15), 0427, Sent by function key f15
key_f16	kf16	F6	KEY_F(16), 0430, Sent by function key f16
key_f17	kf17	F7	KEY_F(17), 0431, Sent by function key f17
key_f18	kf18	F8	KEY_F(18), 0432, Sent by function key f18
key_f19	kf19	F9	KEY_F(19), 0433, Sent by function key f19
key_f20	kf20	FA	KEY_F(20), 0434, Sent by function key f20
key_f21	kf21	FB	KEY_F(21), 0435, Sent by function key f21
key_f22	kf22	FC	KEY_F(22), 0436, Sent by function key f22

key_f23	kf23	FD	KEY_F(23), 0437, Sent by function key f23
key_f24	kf24	FE	KEY_F(24), 0440, Sent by function key f24
key_f25	kf25	FF	KEY_F(25), 0441, Sent by function key f25
key_f26	kf26	FG	KEY_F(26), 0442, Sent by function key f26
key_f27	kf27	FH	KEY_F(27), 0443, Sent by function key f27
key_f28	kf28	FI	KEY_F(28), 0444, Sent by function key f28
key_f29	kf29	FJ	KEY_F(29), 0445, Sent by function key f29
key_f30	kf30	FK	KEY_F(30), 0446, Sent by function key f30
key_f31	kf31	FL	KEY_F(31), 0447, Sent by function key f31
key_f32	kf32	FM	KEY_F(32), 0450, Sent by function key f32
key_f33	kf33	FN	KEY_F(13), 0451, Sent by function key f13
key_f34	kf34	FO	KEY_F(34), 0452, Sent by function key f34
key_f35	kf35	FP	KEY_F(35), 0453, Sent by function key f35
key_f36	kf36	FQ	KEY_F(36), 0454, Sent by function key f36
key_f37	kf37	FR	KEY_F(37), 0455, Sent by function key f37
key_f38	kf38	FS	KEY_F(38), 0456, Sent by function key f38
key_f39	kf39	FT	KEY_F(39), 0457, Sent by function key f39
key_f40	kf40	FU	KEY_F(40), 0460, Sent by function key f40
key_f41	kf41	FV	KEY_F(41), 0461, Sent by function key f41
key_f42	kf42	FW	KEY_F(42), 0462, Sent by function key f42
key_f43	kf43	FX	KEY_F(43), 0463, Sent by function key f43
key_f44	kf44	FY	KEY_F(44), 0464, Sent by function key f44
key_f45	kf45	FZ	KEY_F(45), 0465, Sent by function key f45
key_f46	kf46	Fa	KEY_F(46), 0466, Sent by function key f46
key_f47	kf47	Fb	KEY_F(47), 0467, Sent by function key f47
key_f48	kf48	Fc	KEY_F(48), 0470, Sent by function key f48
key_f49	kf49	Fd	KEY_F(49), 0471, Sent by function key f49

key_f50	kf50	Fe	KEY_F(50), 0472, Sent by function key f50
key_f51	kf51	Ff	KEY_F(51), 0473, Sent by function key f51
key_f52	kf52	Fg	KEY_F(52), 0474, Sent by function key f52
key_f53	kf53	Fh	KEY_F(53), 0475, Sent by function key f53
key_f54	kf54	Fi	KEY_F(54), 0476, Sent by function key f54
key_f55	kf55	Fj	KEY_F(55), 0477, Sent by function key f55
key_f56	kf56	Fk	KEY_F(56), 0500, Sent by function key f56
key_f57	kf57	Fl	KEY_F(57), 0501, Sent by function key f57
key_f58	kf58	Fm	KEY_F(58), 0502, Sent by function key f58
key_f59	kf59	Fn	KEY_F(59), 0503, Sent by function key f59
key_f60	kf60	Fo	KEY_F(60), 0504, Sent by function key f60
key_f61	kf61	Fp	KEY_F(61), 0505, Sent by function key f61
key_f62	kf62	Fq	KEY_F(62), 0506, Sent by function key f62
key_f63	kf63	Fr	KEY_F(63), 0507, Sent by function key f63
key_find	kfnd	@0	KEY_FIND, 0552, Sent by find key
key_help	khlp	%1	KEY_HELP, 0553, Sent by help key
key_home	khome	kh	KEY_HOME, 0406, Sent by home key
key_ic	kich1	kI	KEY_IC, 0513, Sent by ins-char/enter ins-mode key
key_il	kil1	kA	KEY_IL, 0511, Sent by insert-line key
key_left	kcub1	kl	KEY_LEFT, 0404, Sent by terminal left-arrow key
key_ll	kll	kH	KEY_LL, 0533, Sent by home-down key
key_mark	kmrk	%2	KEY_MARK, 0554, Sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, 0555, Sent by message key
key_move	kmov	%4	KEY_MOVE, 0556, Sent by move key
key_next	knxt	%5	KEY_NEXT, 0557, Sent by next-object key
key_npage	knp	kN	KEY_NPAGE, 0522, Sent by next-page key
key_open	kopn	%6	KEY_OPEN, 0560, Sent by open key
key_options	kopt	%7	KEY_OPTIONS, 0561, Sent by options key
key_ppage	kpp	kP	KEY_PPAGE, 0523, Sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, 0562, Sent by previous-object key
key_print	kprt	%9	KEY_PRINT, 0532, Sent by print or copy key

key_redo	krdo	%0	KEY_REDO, 0563, Sent by redo key
key_reference	kref	&1	KEY_REFERENCE, 0564, Sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, 0565, Sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, 0566, Sent by replace key
key_restart	krst	&4	KEY_RESTART, 0567, Sent by restart key
key_resume	kres	&5	KEY_RESUME, 0570, Sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, 0405, Sent by terminal right-arrow key
key_save	ksav	&6	KEY_SAVE, 0571, Sent by save key
key_sbeg	kBEG	&9	KEY_SBEG, 0572, Sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, 0573, Sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, 0574, Sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, 0575, Sent by shifted copy key
key_screate	kCRT	*3	KEY_SCREATE, 0576, Sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, 0577, Sent by shifted delete-char key
key_sdl	kDL	*5	KEY_SDL, 0600, Sent by shifted delete-line key
key_select	kslt	*6	KEY_SELECT, 0601, Sent by select key
key_send	kEND	*7	KEY_SEND, 0602, Sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, 0603, Sent by shifted clear-line key
key_sexit	kEXT	*9	KEY_SEXIT, 0604, Sent by shifted exit key
key_sf	kind	kF	KEY_SF, 0520, Sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, 0605, Sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, 0606, Sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, 0607, Sent by shifted home key
key_sic	kIC	#3	KEY_SIC, 0610, Sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, 0611, Sent by shifted left-arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, 0612, Sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, 0613, Sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, 0614, Sent by shifted next key

key_soptions	kOPT	%d	KEY_SOPTIONS, 0615, Sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, 0616, Sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, 0617, Sent by shifted print key
key_sr	kri	kR	KEY_SR, 0521, Sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, 0620, Sent by shifted redo key
key_sreplace	kRPL	%h	KEY_SREPLACE, 0621, Sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, 0622, Sent by shifted right-arrow key
key_srsume	kRES	%j	KEY_SRSUME, 0623, Sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, 0624, Sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, 0625, Sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, 0524, Sent by set-tab key
key_sundo	kUND	!3	KEY_SUNDO, 0626, Sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, 0627, Sent by suspend key
key_undo	kund	&8	KEY_UNDO, 0630, Sent by undo key
key_up	kcuu1	ku	KEY_UP, 0403, Sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of "keypad-transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad-transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off " meta mode"
meta_on	smm	mm	Turn on " meta mode" (8th bit)
newline	nel	nw	Newline (behaves like **cr** followed by **lf**)
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (G*)
parm_delete_line	dl	DL	Delete #1 lines (G*)
parm_down_cursor	cud	DO	Move cursor down #1 lines. (G*)
parm_ich	ich	IC	Insert #1 blank chars (G*)
parm_index	indn	SF	Scroll forward #1 lines. (G)
parm_insert_line	il	AL	Add #1 new blank lines (G*)

parm_left_cursor	cub	LE	Move cursor left #1 spaces (G)
parm_right_cursor	cuf	RI	Move cursor right #1 spaces. (G*)
parm_rindex	rin	SR	Scroll backward #1 lines. (G)
parm_up_cursor	cuu	UP	Move cursor up #1 lines. (G*)
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times (G*)
req_for_input	rfi	RF	Send next input char (for ptys)
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Vertical position absolute (G)
save_cursor	sc	sc	Save cursor position.
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
set_attributes	sgr	sa	Define the video attributes #1-#9 (G)
set_left_margin	smgl	ML	Set soft left margin
set_right_margin	smgr	MR	Set soft right margin
set_tab	hts	st	Set a tab in all rows, current column.
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4 (G)
tab	ht	ta	Tab to next 8 space hardware tab stop.
to_status_line	tsl	ts	Go to status line, col #1 (G)
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character

SAMPLE ENTRY

The following entry, which describes the *Concept*-100 terminal, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100 | c100| concept | c104 | c100-4p | concept 100,
    am, db, eo, in, mir, ul, xenl,
    cols#80, lines#24, pb#9600, vt#8,
    bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>,
    cnorm=\Ew, cr=^M$<9>, cub1=^H, cud1=^J,
    cuf1=\E=, cup=\Ea%p1%' '%+%c%p2%' '%+%c,
    cuu1=\E;, cvvis=\EW, dch1=\E^A$<16*>, dim=\EE,
    dl1=\E^B$<3*>, ed=\E^C$<16*>, el=\E^U$<16>,
    flash=\Ek$<20>\EK, ht=\t$<8>, il1=\E^R$<3*>,
    ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\0\Eo&\0\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
    rev=\ED, rmcup=\Ev\s\s\s\s$<6>\Ep\r\n,
    rmir=\E\0, rmkx=\Ex, rmso=\Ed\Ee, rmul=\Eg,
    rmul=\Eg, sgr0=\EN\0, smcup=\EU\Ev\s\s8p\Ep\r,
    smir=\E^P, smkx=\EX, smso=\EE\ED, smul=\EG,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Lines beginning with "#" are taken as comment lines. Capabilities in *terminfo* are of three types: boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or particular features, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. 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 **cols**, which indicates the number of columns the terminal has, gives the value **80** for the *Concept*. The value may be specified in decimal, octal or hexadecimal using normal C conventions.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the two- to five-character capname, an '=', and then a string ending at the next following comma. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el=\EK$<3>**, and padding characters are supplied by **tputs**( ) (see *curses*(3X)) to provide this delay. The delay can be either a number, e.g., **20**, or a number followed by an '*' (i.e., **3***), a '/' (i.e., **5/**), or both (i.e., **10*/**). 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. (In the case of insert character, the factor is still the number of lines affected. This is always one unless the terminal has **in** and the software uses it.) 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. (Only one decimal place is allowed.) A '/' indicates that the padding is mandatory. Otherwise, if the terminal has **xon** defined, the padding information is advisory and will only be used for cost estimates or when the terminal is in raw mode. Mandatory padding will be transmitted regardless of the setting of **xon**.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, ^*x* maps to a control-*x* for any appropriate *x*, and the sequences **\n, \l, \r, \t, \b, \f,** and **\s** give a newline, linefeed, return, tab, backspace, formfeed, and space, respectively. Other escapes include: \^ for caret (^); \\ for backslash (\); \, for comma (,); \: for colon (:); and **\0** for null. (**\0** will actually produce **\200**, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a backslash (e.g., \123).

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above. Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Preparing Descriptions

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with **vi**(1) to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **terminfo** file to describe it or the inability of **vi**(1) to work with that terminal. To test a new terminal description, set the environment variable **TERMINFO** to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in */usr/lib/terminfo*. To get the padding for insert-line correct (if the terminal manufacturer did not document it) a severe test is to comment out **xon**, edit a large file at 9600 baud with **vi**(1), delete 16 or so lines from the middle of the screen, then hit the **u** key several times quickly. If the display is corrupted, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal has a screen, then the number of lines on the screen is given by the **lines** 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, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hard-copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**. If the terminal uses the xon-xoff flow-control protocol, like most terminals, specify **xon**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over; for example, you would not normally use "**cuf1**=\s" because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and should never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and screen terminals. Thus the model 33 teletype is described as

33|tty33|tty|model 33 teletype, bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,

while the Lear Siegler ADM-3 is described as

adm3|lsi adm3, am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J, ind=^J, lines#24,

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with **printf**(3S)-like escapes (%**x**) in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special % codes to manipulate it in the manner of a Reverse Polish Notation (postfix) calculator. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary. Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use %**gx**%{**5**}%–.

The % encodings have the following meanings:

%%	outputs '%'
%[[:]*flags*][*width*[.*precision*]][**doxXs**]	as in printf, flags are [–+#] and space
%c	print pop() gives %c
%p[1-9]	push i^{th} parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'*c*'	push char constant *c*
%{*nn*}	push decimal constant *nn*
%l	push strlen(pop())
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%A %O	logical operations: and, or
%! %~	unary operations: push(op pop())
%i	(for ANSI terminals) add 1 to first parm, if one parm present, or first two parms, if more than one parm present
%? expr %t thenpart %e elsepart %;	if-then-else, %e elsepart is optional; else-if's are possible ala Algol 68: %? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e b_5 %; c_i are conditions, b_i are bodies.

If the "–" flag is used with "%[doxXs]", then a colon (:) must be placed between the "%" and the "–" to differentiate the flag from the binary "%–" operator, .e.g "%:-16.16s".

Consider the Hewlett-Packard 2645, 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 zero-padded as two digits.

Thus its **cup** capability is "**cup**=\E&a%p2%2.2dc%p1%2.2dY$<6>".

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, "**cup**=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library routines dealing with *terminfo* set tty modes so that tabs are never expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "**cup**=\E=%p1%'\s'%+%c%p2%'\s'%+%c". After sending "\E=", this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the **\EH** sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features on the terminal.)

If the terminal has row or column absolute-cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cup**. If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the Tektronix 4025.

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 **el**. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; 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 **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**.

If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command -- the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

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 indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using *terminfo*. 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 determine the 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". While these are two logically separate attributes (one line versus multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

terminfo can describe 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 **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding 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**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

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 **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode* (see *curses*(3X)), representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse-video plus half-bright is good, or reverse-video alone; however, different users have different preferences on different terminals.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Micro-Term MIME, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking), **bold** (bold or extra-bright), **dim** (dim or half-bright), **invis** (blanking or invisible text), **prot** (protected), **rev** (reverse-video), **sgr0** (turn off all attribute modes), **smacs** (enter alternate-character-set mode), and **rmacs** (exit alternate-character-set mode). Turning on any of these modes singly may or may not turn off other modes. If a command is necessary before alternate character set mode is entered, give the sequence in **enacs** (enable alternate-character-set mode).

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either **0** or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist. (See the example at the end of this section.)

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of either of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. 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. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by **terminfo**. If the **smcup** sequence will not restore the screen after an **rmcup** sequence is output (to the state prior to outputting **rmcup**), specify **nrrmc**.

If your terminal generates underlined characters by using the underline character (with no special codes needed) even though it does not otherwise overstrike characters, then you should give the capability **ul**. For terminals where a character overstriking another leaves both characters on the screen, give the capability **os**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Example of highlighting: assume that the terminal under question needs the following escape sequences to turn on various modes.

tparm parameter	attribute	escape sequence
	none	\E[0m
p1	standout	\E[0;4;7m
p2	underline	\E[0;3m
p3	reverse	\E[0;4m
p4	blink	\E[0;5m
p5	dim	\E[0;7m
p6	bold	\E[0;3;4m
p7	invis	\E[0;8m
p8	protect	not available
p9	altcharset	^O (off) ^N(on)

Note that each escape sequence requires a **0** to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, since this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be \E[**0;3;5m**. The terminal doesn't have *protect* mode, either, but that cannot be simulated in any way, so **p8** is ignored. The *altcharset* mode is different in that it is either **^O** or **^N** depending on whether it is off or on. If all modes were to be turned on, the sequence would be \E[**0;3;4;5;7;8m^N**.

Now look at when different sequences are output. For example, **;3** is output when either **p2** or **p6** is true, that is, if either *underline* or *bold* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

```
sequence     when to output           terminfo translation

\E[0         always                   \E[0
;3           if p2 or p6              %?%p2%p6%|%t;3%;
;4           if p1 or p3 or p6        %?%p1%p3%|%p6%|%t;4%;
;5           if p4                    %?%p4%t;5%;
;7           if p1 or p5              %?%p1%p5%|%t;7%;
;8           if p7                    %?%p7%t;8%;
m            always                   m
^N or ^O     if p9 ^N, else ^O        %?%p9%t^N%e^O%;
```

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%|%t;4%;%?%p5
     5%;%?%p1%p5%|%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O
```

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 Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. 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 **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f63, the codes they send can be given as **kf0, kf1, ..., kf63**. If the first 11 keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed. Further keys are defined above in the capabilities list.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. A string to program their soft-screen labels can be given as **pln**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and

pfx causes the string to be transmitted to the computer. The capabilities **nlab, lw** and **lh** define how many soft labels there are and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**. **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** (see *tput*(1)) to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the terminal has tab stops that can be saved in non-volatile memory, the *terminfo* description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include: **is1**, **is2**, and **is3**, initialization strings for the terminal; **iprog**, the path name of a program to be run to initialize the terminal; and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *terminfo* description. They must be sent to the terminal each time the user logs in and be output in the following order: run the program **iprog**; output **is1**; output **is2**; set the margins using **mgc, smgl** and **smgr**; set the tabs using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of *tput*(1); see *profile*(4).

Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a harder reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals, from */usr/lib/tabset/**; however, the recommended method is to use the initialization and reset strings.) These strings are output by **tput reset**, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs1**, **rs2**, **rs3**, and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed since the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if**.

If there are commands to set and clear margins, they can be given as **mgc** (clear all margins), **smgl** (set left margin), and **smgr** (set right margin).

Delays

Certain capabilities control padding in the *tty*(7) driver. These are primarily needed by hard-copy terminals, and are used by **tput init** to set tty modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** can be used to set the appropriate delay bits to be set in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Status Lines

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The capability **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

Line Graphics

If the terminal has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph name	vt100+ character
arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	-
diamond	`
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n
scan line 1	o
horizontal line	q
scan line 9	s
left tee (├)	t
right tee (┤)	u
bottom tee (┴)	v
top tee (┬)	w
vertical line	x
bullet	~

The best way to describe a new terminal's line graphics set is to add a third column to the above table with the characters for the new terminal that produce the appropriate glyph when the terminal is in the alternate character set mode. For example,

glyph name	vt100+ char	new tty char
upper left corner	l	R
lower left corner	m	F
upper right corner	k	T
lower right corner	j	G
horizontal line	q	,
vertical line	x	.

Now write down the characters left to right, as in "**acsc**=lRmFkTjGq\,x.".

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char, 'x', 10)** is the same as **xxxxxxxxxx.**

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: If the environment variable **CC** exists, all occurrences of the prototype character are replaced with the character in **CC**.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch**, **dialup**, **patch**, and **network**, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to **virtual** terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi**.

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not **^S** and **^Q**, they may be specified with **xonc** and **xoffc**.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. A variation, **mc5p**, takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the *terminfo* model implemented.

Terminals which can not display tilde (˜) characters, such as certain Hazeltine terminals, should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the *Concept* 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie" therefore, to erase standout mode, it is instead necessary to use delete and insert line.

Those Beehive Superbee terminals which do not transmit the escape or control-C characters, should specify **xsb**, indicating that the f1 key is to be used for escape and the f2 key for control-C.

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 **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be canceled by placing *xx*@ to the left of the capability definition, where *xx* is the capability. For example, the entry

```
att4424-2|Teletype 4424    in    display
function group ii,
               rev@,      sgr@,      smul@,
use=att4424,
```

defines an AT&T 4424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/lib/.COREterm/?/*	subset of compiled terminal description database
/usr/lib/tabset/*	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs)

SEE ALSO

curses(3X), printf(3S), term(5).
captoinfo(1M), infocmp(1M), tic(1M), tty(7) in the *System Administrator's Reference Manual* .
tput(1) in the *User's Reference Manual.*
Chapter 10 of the *Programmer's Guide.*

WARNING

As described in the " Tabs and Initialization" section above, a terminal's initialization strings, **is1**, **is2**, and **is3**, if defined, must be output before a *curses*(3X) program is run. An available mechanism for outputting such strings is **tput init** (see *tput*(1) and *profile*(4)).

Tampering with entries in */usr/lib/.COREterm/?/** or */usr/lib/terminfo/?/** (for example, changing or removing an entry) can affect programs such as *vi*(1) that expect the entry to be present and correct. In particular, removing the description for the " dumb" terminal will cause unexpected problems.

NOTE

The *termcap* database (from earlier releases of UNIX System V) may not be supplied in future releases.

NAME

tic - terminfo compiler

SYNOPSIS

tic [−**v**[n]] [−**c**] file

DESCRIPTION

tic translates a *terminfo*(4) file from the source format into the compiled format. The results are placed in the directory */usr/lib/terminfo*. The compiled format is necessary for use with the library routines described in *curses*(3X).

−**vn** (verbose) output to standard error trace information showing *tic*'s progress. The optional integer *n* is a number from 1 to 10, inclusive, indicating the desired level of detail of information. If *n* is omitted, the default level is 1. If *n* is specified and greater than 1, the level of detail is increased.

−**c** only check *file* for errors. Errors in **use**= links are not detected.

file contains one or more *terminfo*(4) terminal descriptions in source format (see *terminfo*(4)). Each description in the file describes the capabilities of a particular terminal. When a **use**=*entry-name* field is discovered in a terminal entry currently being compiled, *tic* reads in the binary from */usr/lib/terminfo* to complete the entry. (Entries created from *file* will be used first. If the environment variable **TERMINFO** is set, that directory is searched instead of */usr/lib/terminfo*.) *tic* duplicates the capabilities in *entry-name* for the current entry, with the exception of those capabilities that explicitly are defined in the current entry.

If the environment variable **TERMINFO** is set, the compiled results are placed there instead of */usr/lib/terminfo*.

FILES

/usr/lib/terminfo/?/* compiled terminal description data base

SEE ALSO

curses(3X), term(4), terminfo(4) in the *Programmer's Reference Manual*.
Chapter 10 in the *Programmer's Guide*.

WARNINGS

Total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

Terminal names exceeding 14 characters will be truncated to 14 characters and a warning message will be printed.

When the −**c** option is used, duplicate terminal names will not be diagnosed; however, when −**c** is not used, they will be.

BUGS

To allow existing executables from the previous release of the UNIX System to continue to run with the compiled terminfo entries created by the new terminfo compiler, cancelled capabilities will not be marked as cancelled within the terminfo binary unless the entry name has a '+' within it. (Such terminal names are only used for inclusion within other entries via a **use**= entry. Such names would not be used for real terminal names.)

For example:

```
4415+nl, kf1@, kf2@, ....

4415+base, kf1=\EOc, kf2=\EOd, ....

4415-nl|4415 terminal without keys,
        use=4415+nl, use=4415+base,
```

The above example works as expected; the definitions for the keys do not show up in the *4415-nl* entry. However, if the entry *4415+nl* did not have a plus sign within its name, the cancellations would not be marked within the compiled file and the definitions for the function keys would not be cancelled within *4415-nl*.

DIAGNOSTICS

Most diagnostic messages produced by *tic* during the compilation of the source file are preceded with the approximate line number and the name of the terminal currently being worked on.

mkdir ... returned bad status
: The named directory could not be created.

File does not start with terminal names in column one
: The first thing seen in the file, after comments, must be the list of terminal names.

Token after a *seek*(2) not NAMES
: Somehow the file being compiled changed during the compilation.

Not enough memory for use_list element
or
Out of memory
: Not enough free memory was available (*malloc*(3) failed).

Can't open ...
: The named file could not be created.

Error in writing ...
: The named file could not be written to.

Can't link ... to ...
: A link failed.

Error in re-reading compiled file ...
: The compiled file could not be read back in.

Premature EOF
: The current entry ended prematurely.

Backspaced off beginning of line
: This error indicates something wrong happened within *tic*.

Unknown Capability - " ..."
: The named invalid capability was found within the file.

Wrong type used for capability " ..."
: For example, a string capability was given a numeric value.

Unknown token type
: Tokens must be followed by '@' to cancel, ',' for booleans, '#' for numbers, or '=' for strings.

" ..." : bad term name
or
Line ...: Illegal terminal name - " ..."
Terminal names must start with a letter or digit
: The given name was invalid. Names must not contain white space or slashes, and must begin with a letter or digit.

" ..." : terminal name too long.
: An extremely long terminal name was found.

" ..." : terminal name too short.
: A one-letter name was found.

" ..." filename too long, truncating to " ..."
: The given name was truncated to 14 characters due to UNIX file name length limitations.

" ..." defined in more than one entry. Entry being used is " ..." .
: An entry was found more than once.

Terminal name " ..." synonym for itself
: A name was listed twice in the list of synonyms.

At least one synonym should begin with a letter.
: At least one of the names of the terminal should begin with a letter.

Illegal character - " ..."
: The given invalid character was found in the input file.

Newline in middle of terminal name
: The trailing comma was probably left off of the list of names.

Missing comma
: A comma was missing.

Missing numeric value
: The number was missing after a numeric capability.

NULL string value
: The proper way to say that a string capability does not exist is to cancel it.

Very long string found. Missing comma?
: self-explanatory

Unknown option. Usage is:
: An invalid option was entered.

Too many file names. Usage is:
: self-explanatory

" ..." non-existant or permission denied
: The given directory could not be written into.

" ..." is not a directory
    self-explanatory

" ..." : Permission denied
    access denied.

" ..." : Not a directory
    tic wanted to use the given name as a directory, but it already exists as a file

SYSTEM ERROR!! Fork failed!!!
    A *fork*(2) failed.

Error in following up use-links. Either there is a loop in the links or they reference non-existant terminals. The following is a list of the entries involved:
    A *terminfo*(4) entry with a **use**=*name* capability either referenced a non-existant terminal called *name* or *name* somehow referred back to the given entry.

NAME

tput - initialize a terminal or query terminfo database

SYNOPSIS

tput [−**T**type] capname [parms ...]

tput [−**T**type] **init**

tput [−**T**type] **reset**

tput [−**T**type] **longname**

DESCRIPTION

tput uses the *terminfo*(4) database to make the values of terminal-dependent capabilities and information available to the shell (see *sh*(1)), to initialize or reset the terminal, or return the long name of the requested terminal type. *tput* outputs a string if the attribute (*cap*ability *name*) is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, *tput* simply sets the exit code (**0** for TRUE if the terminal has the capability, **1** for FALSE if it does not), and produces no output. Before using a value returned on standard output, the user should test the exit code ($?, see *sh*(1)) to be sure it is **0**. (See **EXIT CODES** and **DIAGNOSTICS** below.) For a complete list of capabilities and the *capname* associated with each, see *terminfo*(4).

−**T***type* indicates the *type* of terminal. Normally this option is unnecessary, because the default is taken from the environment variable **TERM.** If −**T** is specified, then the shell variables **LINES** and **COLUMNS** and the layer size (see *layers*(1)) will not be referenced.

capname indicates the attribute from the *terminfo*(4) database.

parms If the attribute is a string that takes parameters, the arguments *parms* will be instantiated into the string. An all numeric argument will be passed to the attribute as a number.

init If the *terminfo*(4) database is present and an entry for the user's terminal exists (see −**T***type*, above), the following will occur: (1) if present, the terminal's initialization strings will be output (**is1**, **is2**, **is3**, **if**, **iprog**), (2) any delays (e.g., newline) specified in the entry will be set in the tty driver, (3) tabs expansion will be turned on or off according to the specification in the entry, and (4) if tabs are not expanded, standard tabs will be set (every 8 spaces). If an entry does not contain the information needed for any of the four above activities, that activity will silently be skipped.

reset Instead of putting out initialization strings, the terminal's reset strings will be output if present (**rs1**, **rs2**, **rs3**, **rf**). If the reset strings are not present, but initialization strings are, the initialization strings will be output. Otherwise, **reset** acts identically to **init**.

longname If the *terminfo*(4) database is present and an entry for the user's terminal exists (see −**T***type* above), then the long name of the terminal will be put out. The long name is the last name in the first line of the terminal's description in the *terminfo*(4) database (see *term*(5)).

EXAMPLES

tput init Initialize the terminal according to the type of terminal in the environmental variable **TERM**. This command should be included in everyone's .profile after the environmental variable **TERM** has been exported, as illustrated on the *profile*(4) manual page.

tput −T5620 reset Reset an AT&T 5620 terminal, overriding the type of terminal in the environmental variable **TERM**.

tput cup 0 0 Send the sequence to move the cursor to row **0**, column **0** (the upper left corner of the screen, usually known as the " home" cursor position).

tput clear Echo the clear-screen sequence for the current terminal.

tput cols Print the number of columns for the current terminal.

tput -T450 cols Print the number of columns for the 450 terminal.

bold=‘tput smso‘

offbold=‘tput rmso‘ Set the shell variables **bold**, to begin stand-out mode sequence, and **offbold**, to end standout mode sequence, for the current terminal. This might be followed by a prompt:
echo " ${bold}Please type in your name: ${offbold}\c"

tput hc Set exit code to indicate if the current terminal is a hardcopy terminal.

tput cup 23 4 Send the sequence to move the cursor to row 23, column 4.

tput longname Print the long name from the *terminfo*(4) database for the type of terminal specified in the environmental variable **TERM**.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/include/curses.h	*curses*(3X) header file
/usr/include/term.h	*terminfo*(4) header file
/usr/lib/tabset/*	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs); for more information, see the "Tabs and Initialization" section of *terminfo*(4)

SEE ALSO

stty (1), tabs (1).
profile(4), terminfo(4) in the *Programmer's Reference Manual.*
Chapter 10 of the *Programmer's Guide.*

EXIT CODES

If *capname* is of type boolean, a value of **0** is set for TRUE and **1** for FALSE.

If *capname* is of type string, a value of **0** is set if the *capname* is defined for this terminal *type* (the value of *capname* is returned on standard output); a value of **1** is set if *capname* is not defined for this terminal *type* (a null value is returned on standard output).

If *capname* is of type integer, a value of **0** is always set, whether or not *capname* is defined for this terminal *type*. To determine if *capname* is defined for this terminal *type*, the user must test the value of standard output. A value of **−1** means that *capname* is not defined for this terminal *type*.

Any other exit code indicates an error; see **DIAGNOSTICS**, below.

DIAGNOSTICS

tput prints the following error messages and sets the corresponding exit codes.

exit code	error message
0	-1 (*capname* is a numeric variable that is not specified in the *terminfo*(4) database for this terminal type, e.g. **tput −T450 lines** and **tput −T2621 xmc**)
1	no error message is printed, see **EXIT CODES**, above.
2	usage error
3	unknown terminal *type* or no *terminfo*(4) database
4	unknown *terminfo*(4) capability *capname*